# CONSTRUCCIÓN DE ESTRUCTURAS ESPACIALES EN GPU

ISRAEL CABAÑAS RUIZ

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID


Trabajo Fin Máster en Programación y Tecnología Software

Convocatoria: Septiembre
Calificación: Matrícula de Honor

10/9/2012

Director:

Pedro Jesús Martín de la Calle

# Autorización de Difusión

ISRAEL CABAÑAS RUIZ

10/9/2012

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Máster: "CONSTRUCCIÓN DE ESTRUCTURAS ESPACIALES EN GPU", realizado durante el curso académico 2011-2012 bajo la dirección de Pedro Jesús Martín de la Calle en el Departamento de Sistemas Informáticos y Computación (SIC), y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

# Resumen en castellano

Ray tracing es una técnica de renderizado de imágenes que consigue efectos realistas mediante la simulación de la luz con refracción, reflexión y sombras en una escena con objetos 3D. Para hacer eficiente el cálculo de la primera intersección de cada rayo de luz con los objetos de la escena se emplean estructuras jerárquicas para descartar objetos en esa búsqueda. Entre las estructuras jerárquicas, una de las más comunes es el kd-tree, un tipo de estructura arbórea que divide el espacio en regiones, y cuyos nodos engloban regiones que se dividen en sus hijos.

La construcción de kd-trees suele hacerse de manera "top-down". Se parte del nodo raíz que contiene todos los objetos de la escena y se computa un plano que divida la escena en dos partes, una para cada hijo. La búsqueda del plano de corte es clave para crear un kd-tree "óptimo". La mejor medida para encontrarlo es empleando una heurística, la "Surface Area Heuristic" (SAH), que se basa en el cociente de dos superficies, la de la caja que envuelve el nodo y la de la caja de la raíz. En este trabajo se construyen kd-trees en GPU (Graphics Processing Unit) de forma parecida a como han hecho recientemente en Wu et al. [WZL11]. Para construir buenos kd-trees, en ese trabajo se propone el uso de la SAH *exacta* para todos los niveles del árbol, que se caracteriza por tomar como planos candidatos de corte los lados de las cajas que envuelven los objetos del nodo.

El modelo de programación de CUDA permite la creación de programas paralelos de propósito general en GPUs de NVidia. Mientras que las máquinas paralelas en CPU tienen entre 2 o 16 núcleos, en la GPU hay cientos de núcleos, que requieren gran cantidad de trabajo para aprovechar sus recursos. La construcción de kd-trees se puede adaptar a este modelo, consiguiendo tiempos de ejecución tan bajos que permiten su aplicación sobre escenas dinámicas en tiempo real.

# Palabras clave

Ray tracing, Surface Area Heuristic, GPU, Kd-tree, CUDA

# Resumen en inglés

Ray tracing is a rendering technique that achieves realistic effects by simulating light with refraction, reflection and shadows, in a scene with 3D objects. In order to accelerate the computation of the first intersection for each ray of light with the objects in the scene, hierarchical structures are used to discard objects in that search. Among the hierarchical structures, one of the most common is the kd-tree, a tree structure that divides the space into regions, and whose nodes comprise regions that are divided into their children.

The construction of Kd-tree is usually done in a "top-down" manner. It starts from the root node, which contains all the objects in the scene and computes a plane that divides the scene into two parts, one for each child. The search of a cutting plane is the key to create a good kd-tree. The best approach is to use a heuristic, the "Surface Area Heuristic" (SAH), which is based on the ratio of two surfaces, one related to the box enclosing the node and another one related to the root. In this document I present an implementation on GPU (Graphics Processing Unit) for the construction of kd-trees, which is similar to the algorithm proposed by Wu et al. [WZL11]. To build good kd-trees, Wu et al. use the exact SAH for all levels of the tree, which considers as candidate cutting planes the sides of the boxes enclosing the objects.

The CUDA programming model allows the creation of parallel programs for general purpose problems in NVidia's GPUs. While CPU parallel machines have from 2 to 16 cores, in the GPU there are hundreds of cores, which require large amounts of work to exploit their resources. The construction of kd-trees can take advantage of this model, with runtimes so low that its application on dynamic scenes is possible.

# Keywords

Ray tracing, Surface Area Heuristic, GPU, Kd-tree, CUDA

# Índice de contenidos

# Agradecimientos

Mi gratitud, a Dios, a mi familia y amigos, por sus diversas formas de apoyo.

A mi profesor director y los miembros de su grupo de investigación, que con su guía, aportes y material proporcionado, me han sido de gran utilidad para terminar este trabajo.

# Capítulo 1 - KD-Trees

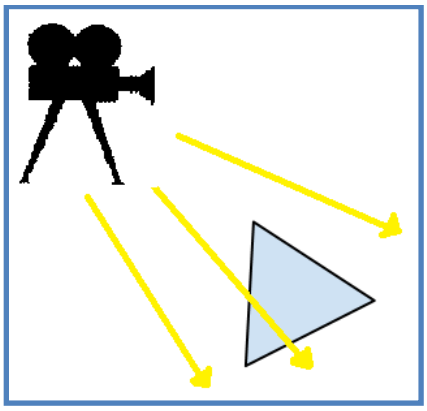
## Ray Tracing

"Ray tracing" es un algoritmo de renderizado de imágenes que sirve para lograr un aspecto realista. Consiste en simular la interacción de la luz en una escena determinando reflexión, refracción y sombras para establecer los colores de los píxeles. La versión simple de este algoritmo lanza rayos desde la cámara a los objetos que componen la escena, siendo de interés saber con cuáles choca cada rayo. La Figura 1.1 muestra el resultado de aplicar esta técnica a una escena 3D en la que se aprecian brillos, sombras, reflejos y profundidad.

**Figura 1.1 Resultado de aplicar "ray tracing" a una escena 3D**

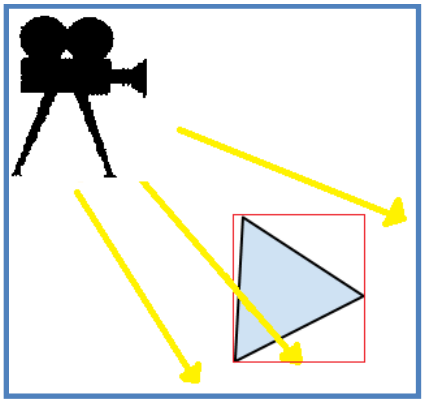
En un primer enfoque se comprobaba la intersección de cada rayo con cada objeto teniendo en cuenta la geometría del mismo, por ejemplo en la Figura 1.2 tres rayos son lanzados contra un triángulo. En ese caso, hay que comprobar para cada rayo la intersección con el triángulo, es decir, averiguar si el píxel correspondiente al rayo en la imagen cae dentro de los tres lados. Cuanto más compleja es la figura geométrica más complejo es el cálculo de intersección, y en consecuencia, mucho mayor el coste del algoritmo de "ray tracing".

**Figura 1.2 Tres rayos lanzados frente a un triángulo**



Un método más eficiente es el uso de "axis aligned bounding boxes" (AABB), cajas alineadas con los ejes de coordenadas, que envuelven al objeto, como por ejemplo la línea roja que envuelve al triángulo de la Figura 1.3. Primero se calcula la intersección con la caja, comparando los límites de la caja con las coordenadas del rayo. Únicamente si el rayo interseca con la caja, se hace el cálculo de intersección con el triángulo, que es un cálculo más complejo. Teniendo en cuenta los tres rayos, en la Figura 1.2 del enfoque anterior había que hacer tres cálculos de intersección contra el triángulo, mientras que ahora en la Figura 1.3 habría que hacer sólo tres cálculos de intersección contra la caja (muy baratos), y el cálculo adicional frente al triángulo para el rayo que interseca la caja.

**Figura 1.3 Tres rayos lanzados frente a un triángulo con una AABB**



En "ray tracing", el algoritmo de fuerza bruta consiste en comprobar para cada rayo la intersección con cada objeto de la escena, se aplique o no el test previo para las AABBs. Para mejorarlo se utiliza la idea de localidad espacial de objetos, y reducir así los cálculos de intersección que se necesitan. De ahí surgió el uso de estructuras arbóreas para dividir el espacio

de la escena, organizando los objetos según su localización en ella de forma jerárquica. De este modo, no es necesario comprobar la intersección contra todos los objetos de la escena, sino que se utiliza la jerarquía para descartar algunas regiones. Entre los tipos de estructuras arbóreas utilizadas con mayor frecuencia están los grids, quadtrees /octrees, kd-trees, Bvhs …

## Definición de kd-tree

El particionado del espacio por estructuras arbóreas divide una región en varias partes, que pueden ser divididas en otras partes más pequeñas sucesivamente. Para entender los kd-trees es necesario comprender primero cómo funcionan los grids y los quadtrees/octrees. Quadtrees y octrees son semejantes, salvo que cuando se trabaja en dos dimensiones se usan quadtrees (cada nodo despliega cuatro hijos) y en tres dimensiones octrees (cada nodo despliega ocho hijos).

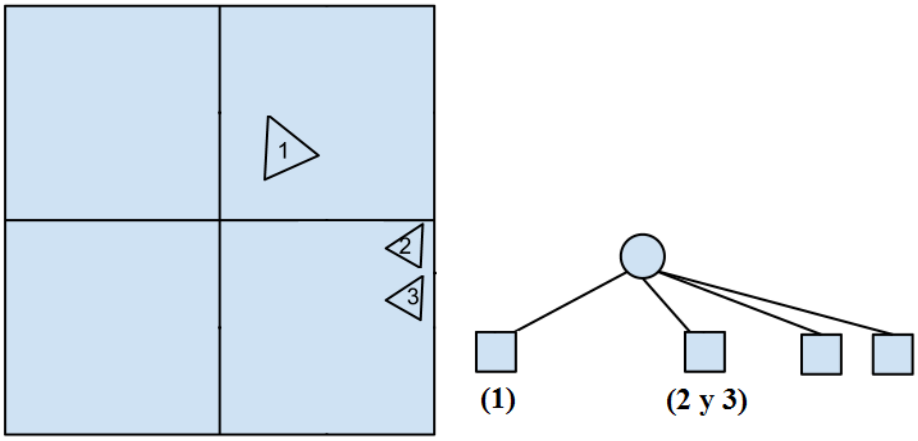
A continuación se describen estas estructuras para facilitar la comprensión de kd-trees, y por último se explica la construcción de kd-trees.

### *Quadtrees*

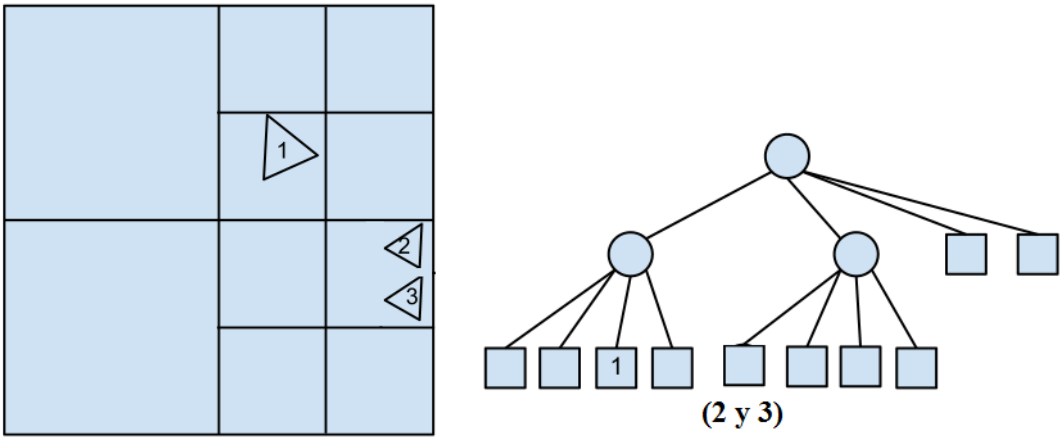
Los quadtrees son árboles cuyos nodos están asociados con cuadrantes. Si un nodo no es una hoja, su cuadrante se parte en cuatro cuadrantes iguales, que son sus hijos. El orden de los cuadrantes para los hijos sigue el sentido horario: superior-derecho, inferior-derecho, inferior-izquierdo y superior-izquierdo. Por ejemplo el árbol de la Figura 1.4 corresponde al particionado de la escena que tiene junto a él. La primera división (a) es la del nodo raíz, que abarca la escena completa. La escena se divide en 4 cuadrantes iguales, donde dos de ellos son hojas vacías y los otros dos son nodos internos. El cuadrante superior-derecho contiene al triángulo 1, y el inferior-derecho a los triángulos 2 y 3. La siguiente división (b) sólo divide aquellos cuadrantes con triángulos. El triángulo 1 queda contenido en el hijo del cuadrante inferior-izquierdo de su nodo, mientras que los triángulos 2 y 3 están juntos en el cuadrante superior-derecho del suyo. La última división (c), expande el único nodo con más de un triángulo, con el triángulo 2 en el cuadrante superior-derecho y el 3 en el inferior-derecho.

El problema de estos árboles es que no estén balanceados y que degeneren en una lista, obligando a atravesar muchos nodos al aplicar “ray tracing” como sucede en el ejemplo de la Figura 1.4, que crea tres niveles de nodos internos para sólo tres triángulos.

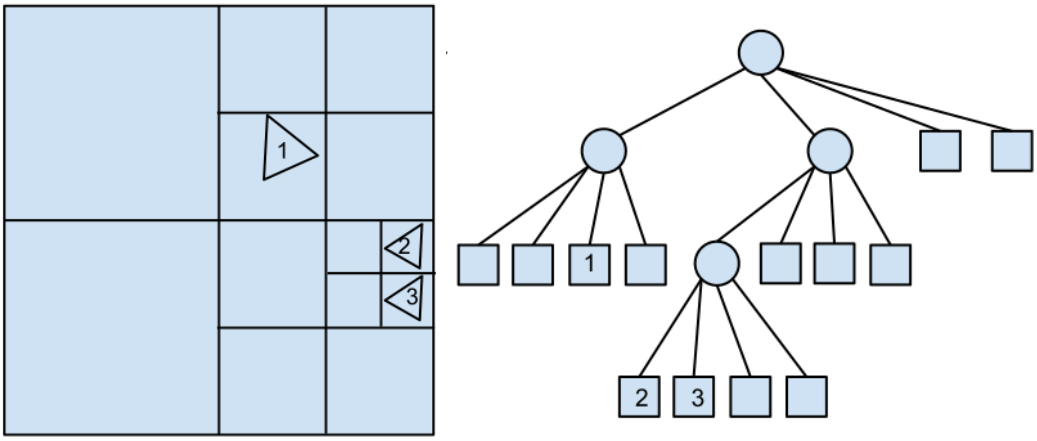
**Figura 1.4 Ejemplo de división de las regiones de una escena con un quadtree.**




(a)




(b)

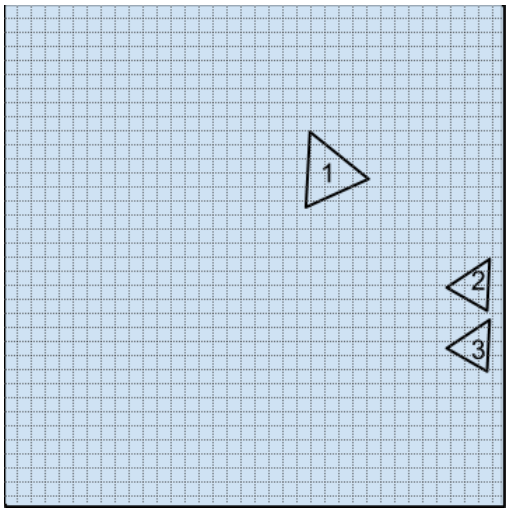



(c)

### *Grids*

Para esta estructura, se divide la escena en una rejilla regular, como en el ejemplo de la Figura 1.5, y para cada celda se guarda una lista con los objetos que la ocupan. Después, según las celdas por las que viaje el rayo, se comprueba la intersección con los objetos de las listas de esas celdas. Hay que tener en cuenta no chequear intersección contra un triángulo ya calculado. Para ello, se puede emplear un sistema de "mailboxes" que guarda el id de los rayos chequeados para cada triángulo.

**Figura 1.5 Ejemplo de particionado de una escena con un grid**



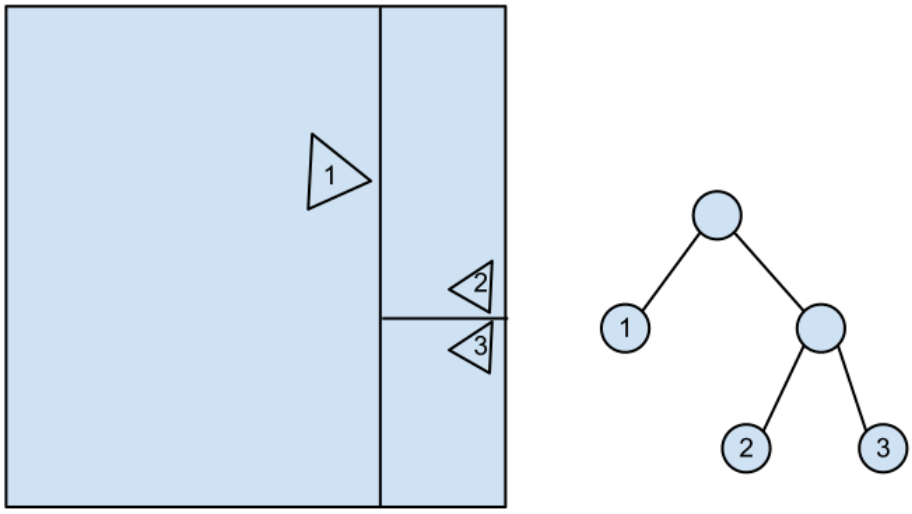

### *Kd-tree*

Un kd-tree es un árbol binario en el que cada división de una región se parte en dos regiones disjuntas. Cada una de ellas puede dividirse en dos más pequeñas sucesivamente hasta los nodos hoja, donde se guarda el listado de los objetos que intersecan con su región. A diferencia del quadtree la división no tiene por qué generar dos particiones iguales. Para hacer el corte o división, se utiliza una recta (para dos dimensiones) o un plano (para tres dimensiones) alineados con los ejes de coordenadas.

Los kd-trees pueden resolver el problema de los quadtrees desequilibrados. Por ejemplo, en la Figura 1.6 se muestra una escena particionada con un kd-tree y el árbol correspondiente. El árbol generado tiene dos niveles internos, uno menos que en el quadtree que para la misma

escena se construyó en la Figura 1.4. La libertad de elegir la recta de corte facilita la construcción de un árbol equilibrado, tomando vital importancia el modo de elegirla.

**Figura 1.6 Ejemplo de particionado de una escena con un kd-tree**




## Construcción de Kd-trees

Los kd-trees se construyen de forma "top-down": cada nodo contiene una región y los nodo internos se parten en regiones disjuntas usando una recta (para el caso de dos dimensiones) o plano (para tres dimensiones) alineado con los ejes de coordenadas. La posición del eje o plano que divide la región es clave para hacer un árbol equilibrado. Hay diferentes enfoques para la elección de este eje o plano:

- La construcción a mano. Puede dar pobres resultados o ser inviable.
- Basándose en la mediana espacial u objetual. Se trata de coger el límite mínimo y máximo en un eje, y coger el valor medio. Según estudios de Choi et al. [CKL*09] se obtienen kd-trees de baja calidad que ralentizan el proceso de renderizado basado en "ray tracing".
- Elegir entre ejes o planos de corte candidatos basándose en heurísticas. Lo más habitual es emplear la "Surface area heuristic" (SAH), ideada por J. Goldsmith y J. Salmon [GS87] en 1987.

***SAH***

Se trata de escoger una recta o plano de corte entre un conjunto de candidatos. Por ejemplo, en la Figura 1.7, la división de un nodo por una recta de corte resulta en dos nodos con cajas más pequeñas. El conjunto de ejes o planos de corte candidatos para dividir una región se pueden tomar de varias formas, entre ellas, las más empleadas son:

- Binned SAH: rectas equidistantes y paralelas a los ejes (…, X = -1, X = 0, X = 1, X = 2, …) como en el ejemplo de la Figura 1.8.
- Exact SAH: tomando los extremos de las AABB de los objetos, como en el ejemplo de la Figura 1.9.

**Figura 1.7 División de una superficie en dos dimensiones con una recta de corte.**

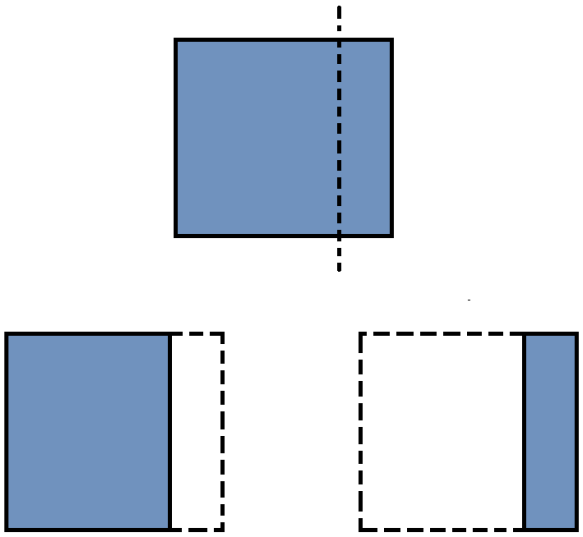


**Figura 1.8 Aplicación de Binned SAH sobre una escena.**

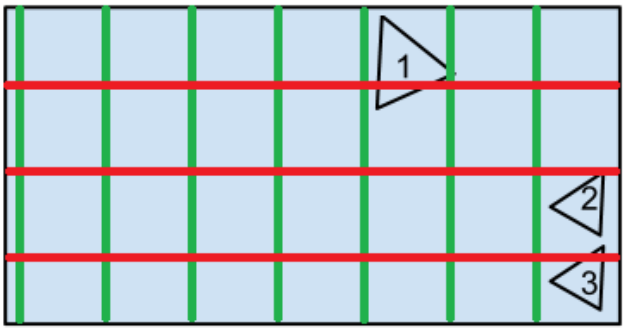
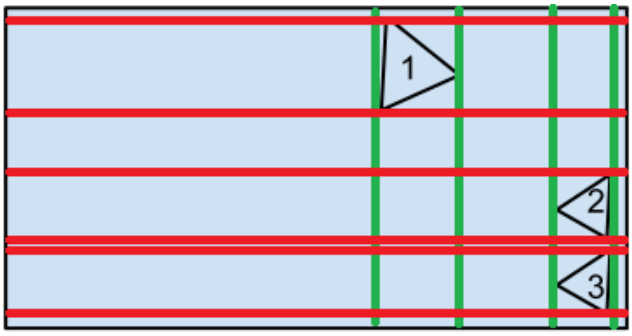


**Figura 1.9 Aplicación de Exact SAH sobre una escena.**




La Figura 1.10 muestra la función de coste recursiva propuesta por J. Goldsmith y J. Salmon [GS87], que mide el coste del árbol cuando está completamente construido, donde x es el nodo sobre el que medir el coste, N(x) es el número de triángulos del nodo hoja y las constantes $C_t$ y $C_i$ los costes de intersección rayo-caja y rayo-primitiva. P(A|B) es la probabilidad de que un rayo interseque la caja de un nodo A, suponiendo que interseca con la caja del nodo B. La construcción "top-down" del árbol que minimiza esta función resulta computacionalmente inviable porque la búsqueda de la recta o plano de corte para un nodo interno requeriría probar todos los candidatos; esto es, construir por completo los árboles óptimos para los 2 hijos que origina cada corte.

**Figura 1.10 Función de coste de SAH.**

$$f(x) = C_t + P(Izq|x) * f(Izq) + P(Der|x) * f(Der)\,, \quad \text{si x es un nodo interno}$$

$$f(x) = C_i * N(x), \quad \text{si x es un nodo hoja}$$

$C_t$: Coste de intersección rayo-caja.

$C_i$: Coste de intersección rayo-primitiva.

P(A|B): Probabilidad de que un rayo interseque con la caja de A suponiendo que interseca con la caja de B.

N(x): Número de primitivas del nodo hoja x.

Respecto al cálculo de las probabilidades, se puede evaluar con probabilidades geométricas. Como muestra la Figura 1.11, la probabilidad de que un rayo interseque con la caja de un nodo A suponiendo que interseca con la caja del nodo B que la contiene se aproxima al cociente de sus

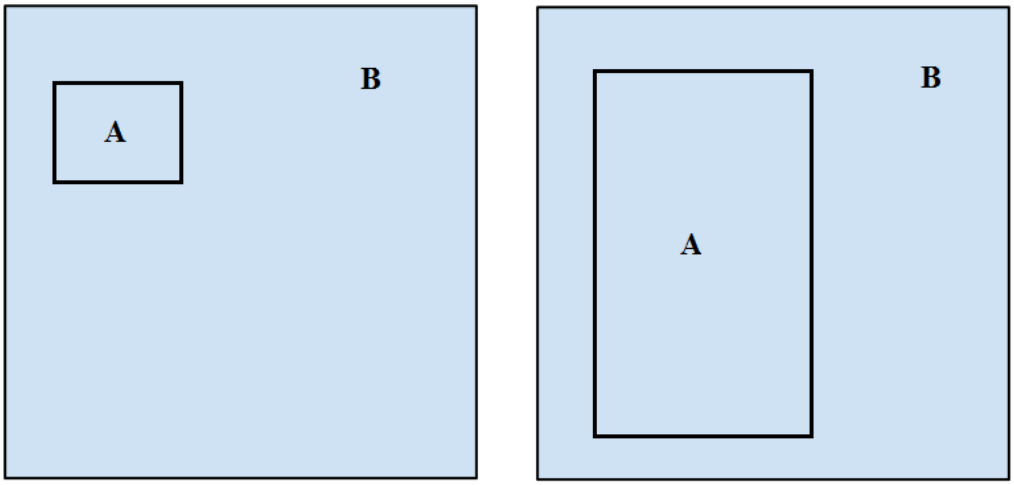
superficies. Por ejemplo, la Figura 1.12 muestra dos cajas A de diferente tamaño dentro de la caja B. Suponiendo que un rayo interseca con la caja B, es menos probable que interseque con la caja A de la imagen de la izquierda que con la caja A de la imagen de la derecha, porque es más pequeña en superficie. Para la aproximación de la Figura 1.11 deben darse tres condiciones en los rayos:

- Todas las direcciones son igualmente probables, es decir, tienen probabilidad constante.
- El origen de cada uno de los rayos está fuera de la escena.
- Los rayos no son bloqueados durante su recorrido, y acaban fuera de la escena.

**Figura 1.11 Aproximación de la probabilidad de intersección de un rayo con la caja del nodo A si interseca con la caja del nodo B que la contiene.**

$$P(A|B) \approx \frac{SA(A)}{SA(B)}$$

**Figura 1.12 Ejemplo para la probabilidad de intersección de un rayo con la caja A suponiendo que interseca con la caja B. En la imagen de la izquierda es menos probable que interseque que en la imagen de la derecha porque tiene menos superficie.**




En 1990, MacDonald y Booth [MB90] simplificaron la función de coste (ver Figura 1.13), eliminando la recursión. La simplificación consiste en tratar al hijo izquierdo y al hijo derecho resultante de un corte candidato como si fueran nodos hoja, de modo que no es necesario

desarrollar los subárboles, si no que es suficiente contar el número de primitivas u objetos que contendrá cada hijo.

Dado un nodo para el que queremos construir el árbol, evaluamos el coste de cada plano de corte candidato x (según SAH exacta o binned SAH) usando la ecuación de la Figura 1.13, donde Izq y Der son los nodos que resultarían de emplear el plano de corte candidato x en el nodo Padre. N(T) es el número de primitivas del nodo T, y SA(T) es la superficie de la caja del nodo T. La constante $C_t$ es el coste de la intersección rayo-caja y $C_i$ el coste de intersección rayo-primitiva. Generalmente el valor empleado para ambas constantes es 1.

**Figura 1.13 Función de coste de SAH simplificada.**

$$f(x) = C_t + C_i * \frac{SA(Izq) * N(Izq) + SA(Der) * N(Der)}{SA(Padre)}$$

$C_t$: Coste de intersección rayo-caja.

$C_i$: Coste de intersección rayo-primitiva.

SA(T): Área de la superficie de la caja del nodo T.

N(T): Número de primitivas dentro de la caja del nodo T.

Una vez seleccionado el eje o el plano de corte con menor valor de SAH, se crean los dos hijos del nodo padre repartiendo sus objetos entre los dos hijos según su localización con respecto a la recta o plano elegido. Los situados a un lado van al hijo izquierdo, y los situados al otro al hijo derecho.

Este proceso de selección de eje o plano y división del nodo se repite hasta que no merezca la pena dividir el nodo. Para decidir esto, existe un umbral que no debe superar la función de coste. Si los valores de SAH de todas las rectas o planos candidatos son mayores que el umbral $f_0$, mostrado en la Figura 1.14, no se hace división y el nodo se convierte en hoja.

**Figura 1.14 Valor umbral de SAH.**

$$f_0 = C_i * N$$

$C_i$: Coste de intersección rayo-primitiva.

N: Número de primitivas del nodo.

# Capítulo 2 - Graphics processing unit (GPU)

Hasta alrededor de 2003 los desarrolladores de software se valían en gran parte del avance de la tecnología del hardware para conseguir que sus aplicaciones secuenciales fueran más rápidas. Desde 2003, ese progreso del hardware se ha reducido a causa del consumo de energía y la disipación de calor que requiere aumentar la velocidad de reloj.

Durante décadas, el desarrollo de programas en paralelo estaba en manos de unos pocos grupos con acceso a máquinas caras de gran escala. Cuando el hardware para programas secuenciales no pudo aumentar su velocidad es cuando proliferaron las máquinas con microprocesadores de varios núcleos, y esto supuso un empuje para la programación paralela.

Existen dos enfoques para este tipo de máquinas paralelas: CPUs multi-núcleo y las unidades de procesamiento gráfico (GPUs). Las CPUs multi-núcleo heredaron de la arquitectura monociclo un diseño orientado a programas secuenciales que se ejecutan concurrentemente. Suelen componerse de 2 a 16 núcleos. Sin embargo, las GPUs se diseñaron para la computación paralela propia del procesamiento de imágenes, por lo que cuentan con un número masivo de unidades de proceso.

La programación de propósito general en GPUs (GPGPU) no tenía herramientas específicas para su desarrollo hasta 2006. Hasta entonces, los desarrolladores empleaban las GPUs usando librerías gráficas (como “HLSL”, “Cg”, “GLSL”) para programar sus aplicaciones. Para usarlas tenían que tratar los datos como imágenes y conocer el funcionamiento de la tubería gráfica.

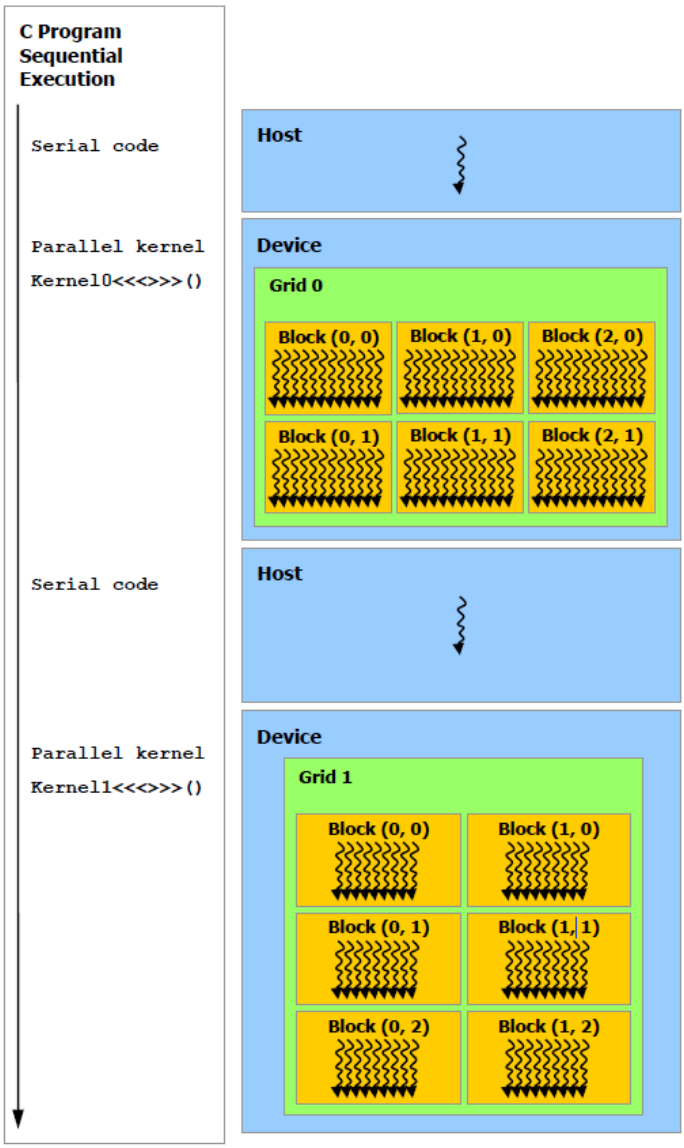
En Noviembre de 2006 nació CUDA. NVidia incluyó hardware en sus tarjetas gráficas específico para el desarrollo de programas de propósito general que no tuvieran que pasar por la tubería gráfica. Además crearon una API para el uso de esta tecnología basado en C/C++.

## Modelo de programación de CUDA

La ejecución de un programa CUDA empieza en el *host* (la CPU tradicional) y cuando un *kernel* (función con el código a ejecutar en la GPU) es invocado, la ejecución se traslada a la GPU. Tanto el *host* como la GPU mantienen cada uno su propia memoria. Es por esta razón que es necesario copiar los datos desde la memoria de host a la memoria global de la GPU antes de invocar el *kernel* que queremos ejecutar en la GPU. La Figura 2.1 muestra el flujo que supone la

ejecución de un programa CUDA en cuanto a la interacción CPU/GPU. Primero se ejecuta código secuencial en la CPU, y al invocar el kernel0, la ejecución se traslada a la GPU para que realice la tarea programada en paralelo. Tras su ejecución, el control vuelve a la CPU que ejecuta más código secuencial, tras el cual se lanza el kernel1 para que la GPU lo ejecute en paralelo.

**Figura 2.1 Flujo de ejecución de una aplicación CUDA (imagen extraída de la guía de programación de CUDA)**




CUDA tiene una función, "cudaMalloc", que permite reservar memoria en la GPU, parecida al tradicional "malloc" de C. En la Figura 2.2 se muestra el código con el que se hace la reserva (líneas 14-16) y se copian los datos (ya existentes en la memoria de la CPU) a la GPU

con otra llamada de CUDA, “cudaMemCpy” (líneas 19 y 20). Tras el copiado de datos se lanza el kernel que opera con ellos en la GPU (línea 23) y a su finalización se copia de vuelta a la memoria de la CPU (línea 26). Por último, igual que hace “free” en C, también debemos liberar memoria en la GPU con “cudaFree” (líneas 28-30) cuando ya no sea necesario usar los datos.

**Figura 2.2 Ejemplo de invocación de un *kernel* en CUDA**

```
00 int main()
01 {
02  // Arrays en el host
03  const int size = 5;
04  const int a[size] = { 1, 2, 3, 4, 5 };
05  const int b[size] = { 10, 20, 30, 40, 50 };
06  int c[size] = { 0 };
07
08    // Punteros para los vectores en la memoria de la GPU
09  int *dev_a = 0;
10  int *dev_b = 0;
11  int *dev_c = 0;
12
13  // Reserva memoria en la GPU para los vectores (2 input, 1 output)    .
14  cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
15  cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
16  cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
17
18  // Copia vectores input de la memoria del host a la memoria de la GPU
19  cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
20  cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
21
22  // Invoca el kernel en la GPU con un hilo para cada elemento
23  addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
24
25  // Copia el vector de salida de la memoria de la GPU a la memoria del host
26     cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
27
28  cudaFree(dev_c);
29  cudaFree(dev_a);
30  cudaFree(dev_b);
31
32  return 0;
33 }
```

Los *kernels* se escriben como funciones de C con algunas características especiales. Van precedidos de “__global__” y se dispone de variables que identifican el hilo que está ejecutando el código, como “threadIdx.x” en la Figura 2.3. Cada hilo ejecuta el código de ese kernel. En él, se realiza la suma de la componente i-ésima de los vectores de entrada (que están en la GPU) y

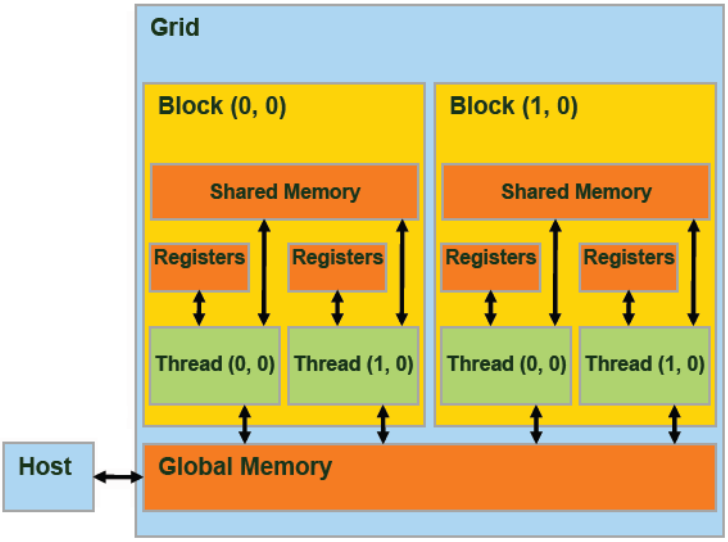
se guarda el resultado en la componente i-ésima en el vector de salida (que también está en la GPU). Cada hilo toma el número de hilo (threadIdx.x) y realiza la suma de la componente i-ésima.

**Figura 2.3 Ejemplo de código de *kernel* en CUDA**

```
00 __global__ void addKernel(int *c, const int *a, const int *b)
01 {
02  int i = threadIdx.x;
03  c[i] = a[i] + b[i];
04 }
```

La invocación de un kernel desde el host tiene dos parámetros obligatorios de entrada rodeados de los separadores "<<< >>>" (ver Figura 2.2, línea 23). Estos son necesarios para especificar un grid. El grid indica cuántos y cómo se organizan los hilos para la invocación del kernel. Se organizan en dos niveles. Primero en un número de *bloques* (agrupaciones de hilos) y segundo, en el número de hilos por bloque. A ambos parámetros se les puede dar estructura bidimensional para organizar los hilos como matrices, si el problema lo requiere. Estos parámetros ayudan al gestor de hilos a establecer la organización de los recursos para la ejecución de un kernel en la GPU. En la Figura 2.2 se opta por un solo bloque con "size" hilos, que es una variable global que depende del tamaño del vector.

**Figura 2.4 Jerarquía de memoria en la GPU (imagen extraída de la guía de programación de CUDA)**



En cuanto a la jerarquía de memoria de la GPU (ver Figura 2.4), cada hilo tiene una memoria local propia (un banco de registros), cada bloque tiene memoria compartida ("shared memory") visible a los hilos del bloque, y por último la memoria global accesible por todos y en la que se copian los datos desde la memoria de la CPU y viceversa con "cudaMemCpy".

La GPU consiste en una serie de multiprocesadores, de forma que cada bloque de hilos se ejecuta en uno de ellos. Cada uno de estos multiprocesadores tiene varias ALUs, memoria compartida, registros para cada hilo y contadores de programa. Las ALUs se agrupan en grupos de 32 para ejecutar la misma instrucción, es decir, hilos asignados a esas ALUs ejecutan la misma instrucción ciclo tras ciclo. A estos grupos de 32 hilos se les llama *warp*. Al ejecutar la misma instrucción se les trata como una unidad de ejecución, por ejemplo, cuando un *warp* está parado esperando ser servido por un acceso a memoria, otro warp toma sus ALUs y así se ocultan latencias.

La capacidad de cómputo de una GPU describe las características soportadas para CUDA, y está definida por dos números que indican su versión. El primero describe la versión del núcleo de la arquitectura, mientras que el segundo indica mejoras al núcleo. Actualmente la versión 2.X (Fermi), es la más extendida. La Tabla 2.1 muestra las características de algunas capacidades de cómputo.

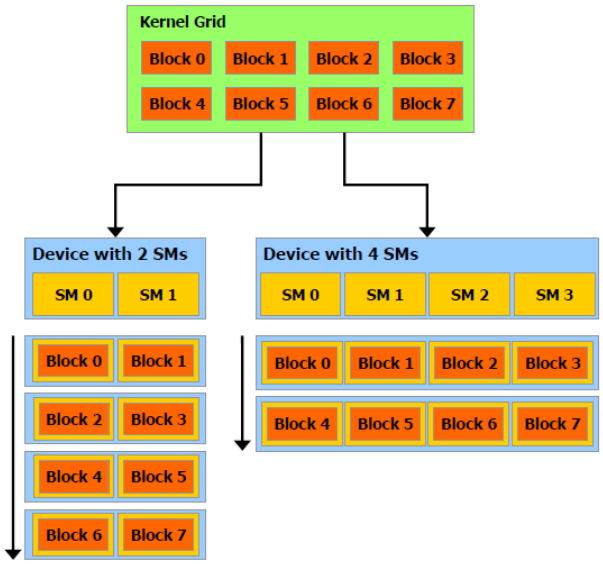
**Tabla 2.1 Soporte de capacidad de cómputo de GPUs.**

<table>
<tr><th></th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th></tr>
<tr><td>Memorial local</td><td colspan="4">16 KB</td><td>512 KB</td></tr>
<tr><td>Memoria compartida por bloque</td><td colspan="4">16 KB</td><td>48 KB</td></tr>
<tr><td>Máx. Nº de hilos por bloque</td><td colspan="4">512</td><td>1024</td></tr>
<tr><td>Operaciones atómicas sobre valores en punto flotante</td><td colspan="4">No</td><td>Sí</td></tr>
<tr><td>Operaciones atómicas sobre enteros</td><td colspan="2">No</td><td colspan="3">Sí</td></tr>
<tr><td>Números en punto flotante de doble precisión</td><td colspan="3">No</td><td colspan="2">Sí</td></tr>
</table>

Este modelo de ejecución está pensado para poder ejecutarse en las tarjetas gráficas de NVidia sin importar las diferencias en el hardware que éstas tengan o de las nuevas versiones de

capacidad de cómputo que aparezcan en el futuro. Cuando un programa CUDA invoca un *kernel* con un grid, los bloques son distribuidos para su ejecución entre los multiprocesadores, donde los hilos de un bloque se ejecutan en paralelo. Por ejemplo, en la Figura 2.5, se muestra como un grid con 8 bloques es distribuido de dos formas para dos GPUs con diferente número de multiprocesadores. En la primera GPU, con 2 multiprocesadores, se ejecutan 2 bloques simultáneamente, mientras que en la segunda GPU, con 4 multiprocesadores se ejecutan 4 bloques al mismo tiempo.

**Figura 2.5 Ejemplo de distribución de bloques para el lanzamiento de un kernel (imagen extraída de la guía de programación de CUDA).**



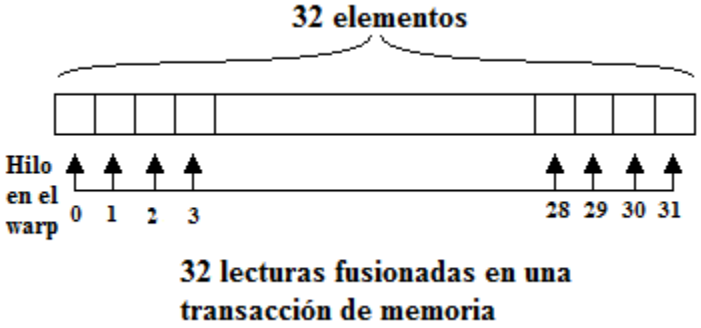

## Optimizando algoritmos en CUDA

En CUDA además de explotar el paralelismo que pueda extraerse del problema a resolver, también se puede sacar partido de la forma de ejecutar los *kernels* en la GPU. En este modelo de ejecución surge un problema con las bifurcaciones o bucles: qué ocurre cuando algunos hilos de un *warp* toman un camino en una bifurcación y los demás hilos del *warp* toman el otro. Esto se contradice con lo comentado anteriormente de que todos los hilos del *warp* tengan que ejecutar la misma instrucción ciclo tras ciclo. La solución que se tomó fue serializar, ejecutando las dos ramas de la bifurcación dejando sin hacer nada a los hilos que no tomen ese camino en la ejecución.

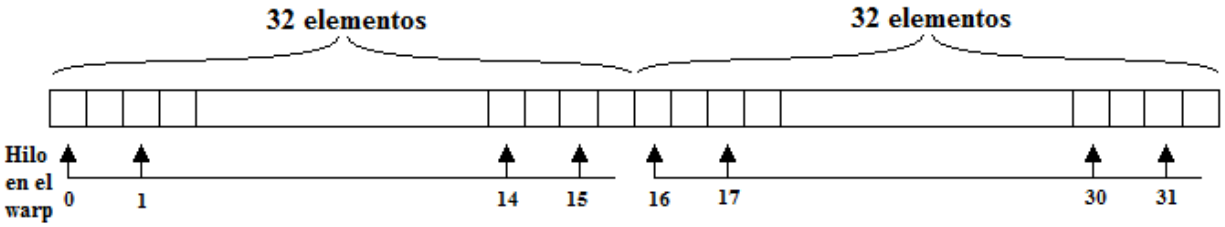
Otro de los problemas que pueden surgir es con los accesos a memoria. Este problema depende de la capacidad de cómputo de la GPU. Actualmente la mayoría de ellas cuentan con 32 hilos en un warp y al realizar un acceso a memoria puede haber tres casos:

- Que pidan los 32 hilos el mismo dato. Este acceso es fusionado, haciendo una única transacción de memoria.
- Aprovechando la localidad espacial de los datos. Si se quiere acceder a 32 datos alineados en un *array*, se realiza una sola transacción porque el controlador de memoria es capaz de leer 32 datos alineados en una única transacción, como en el ejemplo de la Figura 2.6 a).
- Se minimiza el número de transacciones para servir todas las peticiones. Si varias peticiones piden datos de un alineamiento de 32, se fusionan en una única transacción todos aquellos que le son útiles. Como ejemplo, en la Figura 2.6 b) cada hilo del warp pide un elemento par del array, que al no estar alineados, tienen que ser servidos en dos transacciones de memoria.

**Figura 2.6 a) Ejemplo de lecturas con elementos alineados que son fusionadas. b) Ejemplo de lecturas con elementos no alineados y que son fusionadas en 2 transacciones de memoria. c) Ejemplo de lecturas del primer campo de un registro en un array de 32 elementos. d) Lecturas de la primera componente del caso c) si estuvieran alineadas.**

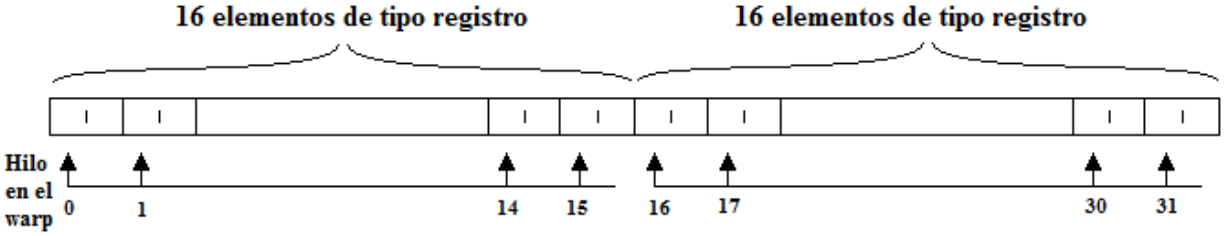



(a)



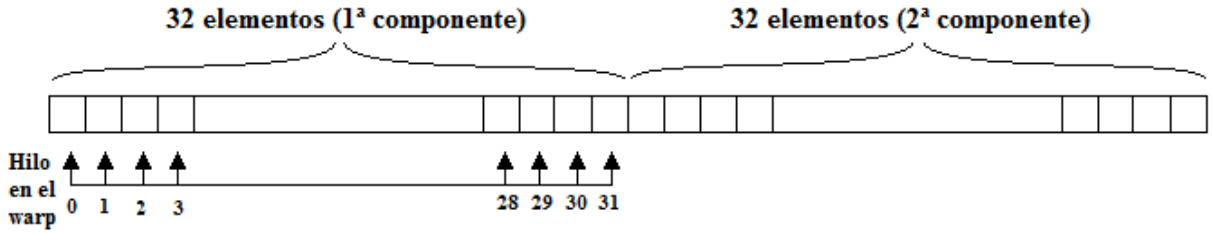

**Lecturas fusionadas en dos transacciones de memoria**

(b)




**Dos transacciones de memoria para leer el primer elemento del registro**

(c)




**Una transacción de memoria para leer la primera componente**

(d)

A la hora de diseñar qué estructuras emplear en los algoritmos, es eficaz tener en cuenta las lecturas fusionadas. La arquitectura CUDA se ve favorecida por emplear registros de arrays, en vez de arrays de registros. Por ejemplo, en la Figura 2.6 c) se emplea un array con registros, 32 elementos de un tipo registro que tiene dos campos de tipo entero, que están almacenados en memoria de forma contigua. Si cada hilo lee el primer campo del registro de su índice se producen lecturas no alineadas, que necesitan dos transacciones de memoria para servirse. Sin embargo, en la Figura 2.6 d) se emplea un registro de arrays, en el que el registro tiene 2 campos

de tipo array. De este modo, se consigue que las primeras componentes estén alineadas en memoria.

Otro aspecto que mejora el tiempo de ejecución es el uso de memoria compartida ("shared memory"). La memoria compartida, que es propia de cada bloque de hilos, no solo sirve para establecer comunicación y un uso común de datos entre los hilos del bloque, sino que también sirve como memoria caché. Los accesos a memoria global tienen una latencia de más de 600 ciclos de reloj. Para la memoria compartida el acceso es de 5-10 ciclos. Una práctica correcta de programación en CUDA consiste en no realizar más de un acceso a memoria global al mismo dato para los hilos de un bloque. Si se necesita usar el dato más de una vez lo mejor es traerlo a memoria compartida y usarlo allí las veces que sea necesario, sin olvidar, llevar el resultado final del *kernel* a memoria global si corresponde.

En resumen, para aprovechar la GPU con CUDA hay que tener en cuenta la localidad de los datos para los accesos a memoria, la divergencia de los hilos de un warp originada por bifurcaciones y bucles, e intentar no realizar más de un acceso a memoria global para el mismo dato dentro de un bloque. De la misma forma, es buena práctica lanzar un número de hilos por bloque que sea múltiplo de 32 para sacar partido a la ejecución en *warps*.

## Primitivas

A lo largo de los últimos años se han desarrollado librerías que implementan en CUDA operaciones habituales sobre arrays. Estas implementaciones aprovechan el modo de funcionamiento del hardware que se ha explicado en el apartado anterior. Entre estas operaciones, denominadas *primitivas*, se encuentran: *scan*, *reduction*, *sort* y versiones segmentadas de estas tres operaciones.

### *Scan*

Scan sirve para computar una suma pre-fija o post-fija de las componentes de un array. Las Figuras 2.6 y 2.7 muestran el resultado de la versión pre-fija para un scan inclusivo y otro exclusivo respectivamente. La mayoría de librerías permiten configurar esta operación para hacerla inclusiva (cada casilla tiene en cuenta para la suma el elemento de su casilla) o exclusiva (lo tienen en cuenta para la siguiente casilla).

**Figura 2.6 Ejemplo de scan inclusivo.**

| In | 1 | 1 | 1 | 0 | 3 | 1 | 0 | 0 | 2 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Out | 1 | 2 | 3 | 3 | 6 | 7 | 7 | 7 | 9 | 11 |

**Figura 2.7 Ejemplo de scan exclusivo.**

| In | 1 | 1 | 1 | 0 | 3 | 1 | 0 | 0 | 2 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Out | 0 | 1 | 2 | 3 | 3 | 6 | 7 | 7 | 7 | 9 |

### *Reduction*

Otra operación habitual es la reducción que consiste en reducir un *array* a un solo elemento. Para ello se especifica la operación que se quiere realizar con cada componente del *array*. Por ejemplo, en la Figura 2.8, se reduce para obtener el mínimo. Se puede utilizar para obtener el máximo, la suma total, producto total. Suele exigirse que la operación sea asociativa.

**Figura 2.8 Ejemplo de reducción para obtener el mínimo.**

| In | 4 | 1 | 1 | 3 | 3 | 1 | 3 | 7 | 2 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Out | 1 | | | | | | | | | |

### *Sort*

Una primitiva de ordenamiento para ordenar las componentes de un *array*.

**Figura 2.9 Ejemplo de ordenamiento de un array aplicando la primitiva sort.**

| In | 4 | 1 | 1 | 3 | 3 | 1 | 3 | 7 | 2 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Out | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 7 |

### *Versiones segmentadas*

Tanto la primitiva scan como la reduction pueden segmentarse. Por ejemplo, en la Figura 2.10, si queremos tratar un *array* (In) con cinco partes y hallar el mínimo de cada una de los cinco segmentos por separado, habría que tener un *array* auxiliar (Segmento) que indicara la pertenencia a cada segmento. Tras realizar la operación, se obtiene el array de salida (Out) con el elemento mínimo de cada parte.

**Figura 2.5 Ejemplo de reducción segmentada con el array de pertenencia al segmento, y el de datos (In). Como resultado el mínimo de cada segmento queda en el array (out).**

| Segmento | 0 | 0 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| In | 4 | 1 | 1 | 3 | 3 | 1 | 3 | 7 | 2 | 2 |

| Segmento | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Out | 1 | 1 | 3 | 1 | 2 |

# Capítulo 3 - Construcción de kd-trees para segmentos (1D)

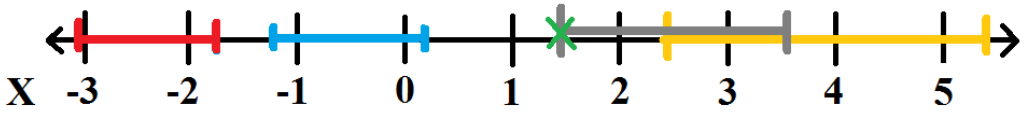
Para hacer un desarrollo cómodo del proyecto de construcción de kd-trees he procedido en tres grandes fases:

- Construcción del árbol para un conjunto de segmentos (una dimensión).
- Construcción para triángulos en dos dimensiones.
- Construcción para triángulos en tres dimensiones.

Este diseño facilita la depuración de los algoritmos y la comprobación de corrección de los resultados. Ya que las implementaciones para dos y tres dimensiones son muy semejantes en cuanto a las estructuras que se precisan, sólo se explican los casos para una y dos dimensiones.

El caso de una dimensión consiste en tener segmentos colocados sobre una recta, como en el ejemplo de la Figura 3.1. A los extremos de los segmentos se les denomina eventos, y se clasifican en start, los menores, y end, los mayores. En la Figura 3.1 se muestran cuatro segmentos (colores rojo, azul, gris y naranja) y un punto de corte (en verde). El punto de corte indica que los segmentos rojo y azul irán al hijo izquierdo del nodo raíz, y los segmentos gris y naranja al hijo derecho.

**Figura 3.1 Ejemplo de segmentos en una recta con un punto de corte señalado en verde.**




## Descripción global del algoritmo

El Algoritmo 3.1 es un esbozo del algoritmo para la construcción del kd-tree, cuyos pasos se detallan en los siguientes apartados. El primer paso, la inicialización (línea 3), consiste en copiar las coordenadas de los segmentos a la GPU, hallar la AABB de cada segmento e inicializar las estructuras del algoritmo. Después continúa con un bucle, cuyo número de iteraciones se predice empleando el número de segmentos a tratar. En este bucle se genera un nivel del árbol por cada iteración. Cada iteración realiza la búsqueda del punto de corte de cada nodo vivo (línea 7), los divide para generar los nodos hijos (línea 9), extrae la información de la GPU necesaria para el próximo nivel (línea 15) e intercambia los arrays sobre los que escribir los

resultados parciales (línea 17). Por último, el resultado es devuelto en un objeto de la clase KDTree (línea 20) extrayendo de la GPU los arrays que contienen el árbol construido.

**Algoritmo 3.1 Código del algoritmo global.**

```
00 KDTree* KDTreeGenerator::genTree()
01 {
02    // Inicialización de estructuras
03    init();
04    // Creación del árbol por niveles
05    for (unsigned int i = 0; i < memPredictor->getNumLevels() - 1; i++) {
06          // Selección de los puntos de corte para cada nodos del nivel
07          bestPlaneSelector->selectBestPlane(events->getActualSwap(), nodes);
08          // Divide los nodos para crear la siguiente generación
09          splitter->split(
10                events->getOffSwap(),
11                events->getActualSwap(),
12                nodes,
13                events->getNumEvents());
14          // Extrae la información para el siguiente nivel
15          memPredictor->nextLevel();
16          // Cambia los arrays de swap sobre los que escribir
17          events->swap();
18    }
19    // Extrae de la GPU el árbol resultante
20    return end();
21 }
```

## Pasos previos

Antes de la generación del árbol, los puntos cargados en la memoria del host desde un archivo son copiados con "cudaMemcpy" en la GPU. Una vez se conoce el número de segmentos, se hacen las siguientes predicciones:

- Cota superior del número de iteraciones / niveles: Existe un determinado crecimiento en el número de referencias a los segmentos de los nodos del árbol debido a los segmentos cruzados por un punto de corte. Cuando un segmento es cruzado por un punto de corte, la parte izquierda pertenece al hijo izquierdo, y la parte derecha al hijo derecho. Esto produce dos referencias al mismo segmento desde dos nodos diferentes (el segmento es duplicado). En consecuencia, el número de niveles no puede aproximarse con facilidad usando $\log_2(numSegmentos)$ directamente.

Havran [WH06] hizo una estimación experimental del número de triángulos que son duplicados por una recta: $\sqrt{numTriangulos}$. Con objeto de construir un árbol balanceado con una referencia por hoja y conocer el número de niveles necesarios, he empleado la siguiente estimación experimentalmente: $\log_2(numSegmentos) + 0.2 * \log_2(numSegmentos)$. La principal ventaja es que antes de construir el árbol se puede estimar la profundidad del mismo y reservar memoria para las estructuras que lo almacenan. En cambio, al ser una estimación, falla en los siguientes casos:

  - Si los segmentos están altamente concentrados, producen un mayor número de duplicados, necesitando más niveles para terminar de construir el árbol.
  - Si los segmentos están dispersos, producen menos duplicados y se necesitan menos niveles para terminar el algoritmo, realizando iteraciones innecesarias.

- Cota superior del número de nodos. Sabiendo el número de iteraciones, reservamos la cantidad de nodos que tendría un árbol binario completo, es decir, $2^{numNiveles} - 1$.
- Cota superior del máximo número de eventos. Durante la construcción por niveles algunos eventos serán duplicados, formando parte del hijo izquierdo y del derecho. Las pruebas realizadas muestran que es suficiente con cuatro veces el número de segmentos.

Dado que la reserva de memoria en GPU es costosa, no conviene reservar y liberar memoria de la GPU durante la ejecución del algoritmo. Con estas cotas superiores se reserva la memoria necesaria para las estructuras del algoritmo al comienzo del mismo y se libera al terminar.

## Inicialización

Esta primera fase de la generación del kd-tree sigue los pasos que muestra el Algoritmo 3.2: hallar las AABB de cada segmento (línea 3), ordenar los eventos, es decir, los límites de todas las AABB (línea 7), e inicializar las estructuras asociadas a eventos (línea 10) y nodos del

árbol (línea 13). Por último se inicializa un array de punteros de modo que cada evento tenga un puntero al evento que corresponde al otro extremo de su segmento (línea 16).

**Algoritmo 3.2 Código para la inicialización.**

```
00    void KDTreeGenerator::init()
01    {
02          // Cálculo de los eventos o bounding boxes por cada primitiva
03          computeAABB();
04
05          // Ordena los eventos en un array y conserva el índice de su
06          // segmento dueño en otro array mapeado con él
07          sortEventLists();
08
09          // Establece el nodo raíz como nodo dueño de todos los eventos
10          initEvents();
11
12          // Inicializa los arrays de nodos con valores UNUSED excepto el nodo raíz
13          initNodes();
14
15          // Inicializa los punteros a la pareja de cada evento
16          initPairPointers();
17    }
```

### *Cálculo de AABB*

El cálculo de una AABB para un segmento es trivial, el Kernel para su cálculo accede a los dos extremos y los ordena, ya que podrán estar desordenados. El Algoritmo 3.3 muestra el kernel que ejecuta un hilo por segmento, en el que se hace uso de memoria compartida para traer el valor de los dos extremos (líneas 18 y 19) y hacer las comparaciones que determinan el mayor y el menor. Solo al final se hacen las escrituras a memoria (líneas 33 y 34) para que todos los hilos del mismo warp que ejecuten esa instrucción hagan la escritura al mismo tiempo, es decir, de forma fusionada. Como entrada cada segmento en d_segments (línea 1) tiene sus dos extremos en el eje X almacenados como se muestra en la Figura 3.2 para el ejemplo de la Figura 3.1. Los primeros extremos de cada segmento se colocan consecutivamente y a continuación los segundos. Esta estructura es idónea para hacer lecturas y escrituras fusionadas por warp. Supongamos que el hilo correspondiente a un segmento toma el primer extremo de ese segmento. Al ocupar posiciones contiguas los accesos se hacen con lecturas fusionadas que se sirven con una única transacción por warp. Sin embargo, si se hubiera tomado una estructura en la que los segundos extremos estuvieran intercalados entre los primeros, estos accesos a memoria

solo tomarían la mitad de los datos en cada transacción, y sería necesario hacer una segunda transacción.

**Figura 3.2 Inicialización para los segmentos de la Figura 3.1.**


**Algoritmo 3.3 Kernel para computar la AABB de cada segmento en paralelo.**

```
00    __global__ void KernelComputeAABB(     float* d_AABB,
01                                            float* d_segments,
02                                            unsigned int numSegments)
03    {
04          // Declaración de arrays en memoria compartida
05          __shared__ float d_segmentsFirstTmp[THREADS_PER_BLOCK];
06          __shared__ float d_segmentsSecondTmp[THREADS_PER_BLOCK];
07
08          // Número de hilo
09          unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
10
11          // Ordena las dos componentes del segmento colocando primero
12          // la menor y luego la mayor para así obtener su bounding box
13          if (i < numSegments) {
14                // Precalculo de las posiciones a ocupar de cada punto
15                unsigned int firstPoint = i;
16                unsigned int secondPoint = firstPoint + numSegments;
17                // Copiado de memoria global a memoria compartida de los puntos del segmento
18                d_segmentsFirstTmp[threadIdx.x] = d_segments[firstPoint];
19                d_segmentsSecondTmp[threadIdx.x] = d_segments[secondPoint];
20                // Ordenación de los dos puntos: primero el menor, después el mayor
21                float lesser;
22                float greater;
23                if (d_segmentsFirstTmp[threadIdx.x] > d_segmentsSecondTmp[threadIdx.x]) {
24                      lesser = d_segmentsSecondTmp[threadIdx.x];
25                      greater = d_segmentsFirstTmp[threadIdx.x];
26                }
27                else {
28                      lesser = d_segmentsFirstTmp[threadIdx.x];
29                      greater = d_segmentsSecondTmp[threadIdx.x];
30                }
31                // Escrituras a memoria global de forma unificada para todos
32                // los hilos del warp
33                d_AABB[firstPoint] = lesser;
34                d_AABB[secondPoint] = greater;
35          }
36    }
```

### *Ordenación de eventos*

Después se crea el array de eventos ordenados (d_eventX). Los eventos son los comienzos en el eje X de las AABBs (eventos start) y los finales (eventos end). Este array mantiene ordenados los eventos de menor a mayor y será vital para el cálculo de la SAH más adelante. Para ordenarlos, se parte del array de AABBs y se emplea la primitiva sort de la librería THRUST, como en el Algoritmo 3.4, que además de devolver el array de eventos ordenado también devuelve la posición que ocupaba cada evento antes de la ordenación. Para ello, el kernel del Algoritmo 3.4 escribe para cada posición del array "índices" el índice de su posición (línea 4), obteniendo un array en el que en la posición cero tiene escrito el valor cero, en la posición uno el valor uno, y así hasta numEvents - 1. En el método sortEventList primero se lanza ese kernel (línea 14), obteniendo en d_presortPosX el array de índices. Ese array se emplea en la llamada a sort_by_key de THRUST de forma que durante la ordenación, además de mover cada evento mueve junto a él el valor en d_presortPosX, es decir, el índice que tenía el evento antes de ordenarse. Ya que el array d_presortPosX contiene para cada evento el índice original que tenía en d_AABB antes de realizar la ordenación, este array sirve para determinar si un evento es start o end comprobando si esa posición es mayor o menor que el número de segmentos inicial, es decir, si estaba en el lado de los primeros extremos o en el de los segundos.

**Algoritmo 3.4 Ordenación de eventos empleando la primitiva sort de THRUST.**

```
00   __global__ void KernelSetIndices(unsigned int* indices, unsigned int numIndices)
01   {
02         int i = threadIdx.x + blockIdx.x * blockDim.x;
03         if (i < numIndices)
04               indices[i] = i;
05   }
06
07   void sortEventLists(
08         float*              d_eventX,
09         unsigned int*   d_presortPosX,
10         float*              d_AABB,
11         unsigned int    numEvents)
12   {
13         unsigned int numBlocks = (numEvents + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
14         KernelSetIndices<<<numBlocks, THREADS_PER_BLOCK>>>(d_presortPosX, numEvents);
15
16         thrust::device_ptr<float> d_thrust_eventX(d_eventX);
17         thrust::device_ptr<unsigned int> d_thrust_presortPosX (d_presortPosX);
18         thrust::sort_by_key(
19               d_thrust_eventX,
20               d_thrust_eventX + numEvents,
21               d_thrust_presortPosX);
22   }
```

### *Inicialización de otras estructuras*

El Algoritmo 3.2 continúa con la inicialización de otras estructuras para eventos y nodos (líneas 10, 13 y 16). Para los eventos hay un array que indica el nodo al que pertenece cada uno y un entero sin signo que indica el número total de eventos contenidos en el array. Inicialmente todos pertenecen al nodo raíz cuyo número es el cero, y el número de eventos es $2 \times numSegments$ porque cada segmento aporta dos eventos.

Para los nodos, los arrays que almacenan sus datos tienen la raíz en la posición cero, los hijos izquierdos van a la posición 2 * i + 1 y los derechos 2 * i + 2. Por ejemplo el hijo izquierdo del nodo cero está en la posición 1 (2 * 0 + 1 = 1) y el hijo derecho en la posición 2 (2 * 0 + 2 = 2). El array tiene capacidad para almacenar un árbol completo. Si algún nodo no tiene hijos, las posiciones de sus hijos quedan sin utilizar.

Para finalizar la inicialización, se escribe en un array, d_pairEvent, el puntero de cada evento a su pareja, es decir, la posición en el array de eventos del otro extremo del segmento del que forma parte. De esta manera cada evento start puede acceder a su evento end, y viceversa. Obtener estos punteros para cada evento requiere emplear un array auxiliar. En el Algoritmo 3.5, en un primer kernel, cada evento escribe en el array auxiliar en el índice original del evento (antes de la ordenación), d_presortPosX (línea 7), la posición que está ocupando en el array de eventos ya ordenado. En un segundo kernel, cada evento lee la posición que dejó escrita su pareja en el array auxiliar. La manera de saber si un evento es un evento start o end es comprobar si su índice original ocupaba una posición superior al número de segmentos de inicio (línea 20). Los eventos end toman el índice que ahora ocupe su evento start (línea 21) y los eventos start el índice de su evento end (línea 23).

**Algoritmo 3.5 Kernels para la obtención de la pareja de cada evento.**

```
00    __global__ void KernelStorePairs(
01          unsigned int* d_aux,
02          unsigned int* d_presortPosX,
03          unsigned int numEvents)
04    {
05          int i = threadIdx.x + blockIdx.x * blockDim.x;
06          if (i < numEvents)
07                d_aux[ d_presortPosX[i] ] = i;
08    }
09
10    __global__ void KernelLoadPairs(
11          unsigned int* d_pairEvent,
12          unsigned int* d_aux,
13          unsigned int* d_presortPosX,
14          unsigned int numSegments,
15          unsigned int numEvents)
16    {
17          int i = threadIdx.x + blockIdx.x * blockDim.x;
18          if (i < numEvents) {
19                unsigned int owner = d_presortPosX[i];
20                if (owner > numSegments)
21                      d_pairEvent[i] = d_aux[owner - numSegments];
22                else
23                      d_pairEvent[i] = d_aux[owner + numSegments];
24          }
25    }
```

### *Ejemplo de inicialización*

La Figura 3.3 muestra la inicialización para el ejemplo de la Figura 3.1. Primero se copian las componentes a la GPU, se calculan las AABBs y se ordenan los eventos, obteniendo la posición que tenían (d_presortPosX) en el array sin ordenar (d_AABB). Al comienzo del algoritmo sólo tenemos como nodo vivo el nodo raíz, nodo cero, por lo tanto el nodo correspondiente a todos los eventos actuales es 0 (d_nodeForEvent). Para la inicialización de nodos se establece que la posición de comienzo del nodo raíz en el array de eventos (d_nodeInitialPos) es la posición 0. También se indica el número de segmentos que contiene el nodo raíz (4), y los límites de la AABB del nodo que coinciden con el primer evento y el último. Por último, en d_pairEvent, cada evento tiene la posición en el array de eventos del evento del otro extremo de su segmento.

**Figura 3.3 Inicialización para los segmentos de la Figura 3.1.**

⇩ Copia de los segmentos a la memoria de la GPU

| d_segments | -1.7 | -1.3 | 3.5 | 2.5 | -3.1 | 0.2 | 1.4 | 5.5 |
|---|---|---|---|---|---|---|---|---|

⇩ computeAABB

| d_AABB | -3.1 | -1.3 | 1.4 | 2.5 | -1.7 | 0.2 | 3.5 | 5.5 |
|---|---|---|---|---|---|---|---|---|

⇩ sortEventLists

| d_eventX | -3.1 | -1.7 | -1.3 | 0.2 | 1.4 | 2.5 | 3.5 | 5.5 |
|---|---|---|---|---|---|---|---|---|
| d_presortPosX | 0 | 4 | 1 | 5 | 2 | 3 | 6 | 7 |

⇩ initEvents

| d_nodeForEvent | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| d_numEvents | 8 | | | | | | | |

⇩ initNodes

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeInitialPos | 0 | | | | | | |
| d_nodeNumSegments | 4 | | | | | | |
| d_nodeAABBXMin | -3.1 | | | | | | |
| d_nodeAABBXMax | 5.5 | | | | | | |

⇩ initPairEvents

| d_pairEvent | 1 | 0 | 3 | 2 | 6 | 7 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|

## Selección de mejores puntos de corte

### *Generación de flags*

Se comienza generando un array de flags en el que cada evento escribe un 1 si es un evento start (es decir, su índice en d_AABB antes de la ordenación está en el rango de primeros extremos) o 0 si es un evento end (su índice está en el rango de segundos extremos). La Figura 3.5 busca para el ejemplo del apartado anterior el punto de corte para el nodo raíz. En la primera etapa de la figura, cada evento escribe su flag comprobando si el índice antes de la ordenación (d_presortPosX), es mayor o menor que el número de segmentos inicial; es decir, si la posición en el array d_AABB es menor que 4, entonces se trata de un evento start, si no, es un evento end.

### *Cálculo de la SAH*

A continuación se emplea la primitiva scan de THRUST para hacer la suma prefija exclusiva de ese array de eventos. El array resultado del scan, "d_scannedFlags", sirve para saber el número de segmentos que caen a cada lado de cada evento. Y es cada evento el que se toma como candidato a punto de corte y sobre el que se calcula la SAH. El kernel del Algoritmo 3.6 es el que realiza el cálculo de SAH por evento. Los datos necesarios para aplicar la función de coste SAH de la Figura 3.4 (explicada en la sección 2) se obtienen de la siguiente forma en el Algoritmo 3.6:

- El número de segmentos a la izquierda del punto de corte candidato "NL" (línea 27 y 36), se obtiene usando los valores almacenados en el array resultado del scan, "d_scannedFlags".
- El número de segmentos a la derecha del punto de corte candidato "NR" (línea 38), empleando el número de segmentos total, la posición en el array de eventos y el array "d_scannedFlags".
- En el caso de una dimensión, en vez de tratar con superficies para el cálculo de SAH, se emplean longitudes. Las longitudes de los hijos resultantes del punto de corte candidato (líneas 37 y 39) se obtienen con la diferencia entre el punto candidato y el primer evento del nodo padre (hijo izquierdo), y la diferencie entre el último evento y el punto de corte (hijo derecho). La longitud del nodo padre

(línea 40) se obtiene como la diferencia entre el último y el primer evento del nodo.

- Los valores de la fórmula de la Figura 3.4 para el coste de intersección rayo-caja ($C_t$) y rayo-primitiva ($C_i$) se han tomado como 1.

Solo en caso de que el número de primitivas del nodo supere un umbral (línea 20 ó 42), o que se intente tomar un borde de la caja del padre como candidato (línea 30), se marca la SAH con un coste elevado (valor UNUSED).

**Figura 3.4 Función de coste de SAH simplificada.**

$$f(x) = C_t + C_i * \frac{SA(Izq) * N(Izq) + SA(Der) * N(Der)}{SA(Padre)}$$

$C_t$: Coste de intersección rayo-caja.

$C_i$: Coste de intersección rayo-primitiva.

SA(T): Área de la superficie de la caja del nodo T.

N(T): Número de primitivas dentro de la caja del nodo T.

En el ejemplo de la Figura 3.5, para el evento 2.5 (posición 5 en d_eventX), d_scannedFlags tiene un valor de 3, por lo que el valor de NL es 3 (línea 36 del Algoritmo 3.6). Para calcular NR (línea 38), requiere el número de segmentos del nodo (4), la posición del evento (5), la posición del primer evento del nodo (0) y el valor de d_scannedFlags (3); obteniéndose un valor de 2 (4 – (5 – 0 – 3) = 2). La longitud del padre se obtiene con la diferencia entre el mayor y menor evento del nodo, (5.5 – (-3.1) = 8.6). La longitud del hijo izquierdo se calcula como la diferencia entre el punto de corte y el menor evento del nodo (2.5 – (-3.1) = 5.6) y la del hijo derecho, la diferencia entre el mayor evento y el punto de corte (5.5 – 2.5 = 3). Al aplicar la fórmula se obtiene como resultado 3.65, el valor escrito en la posición 5 de d_sahEvents en la Figura 3.5.

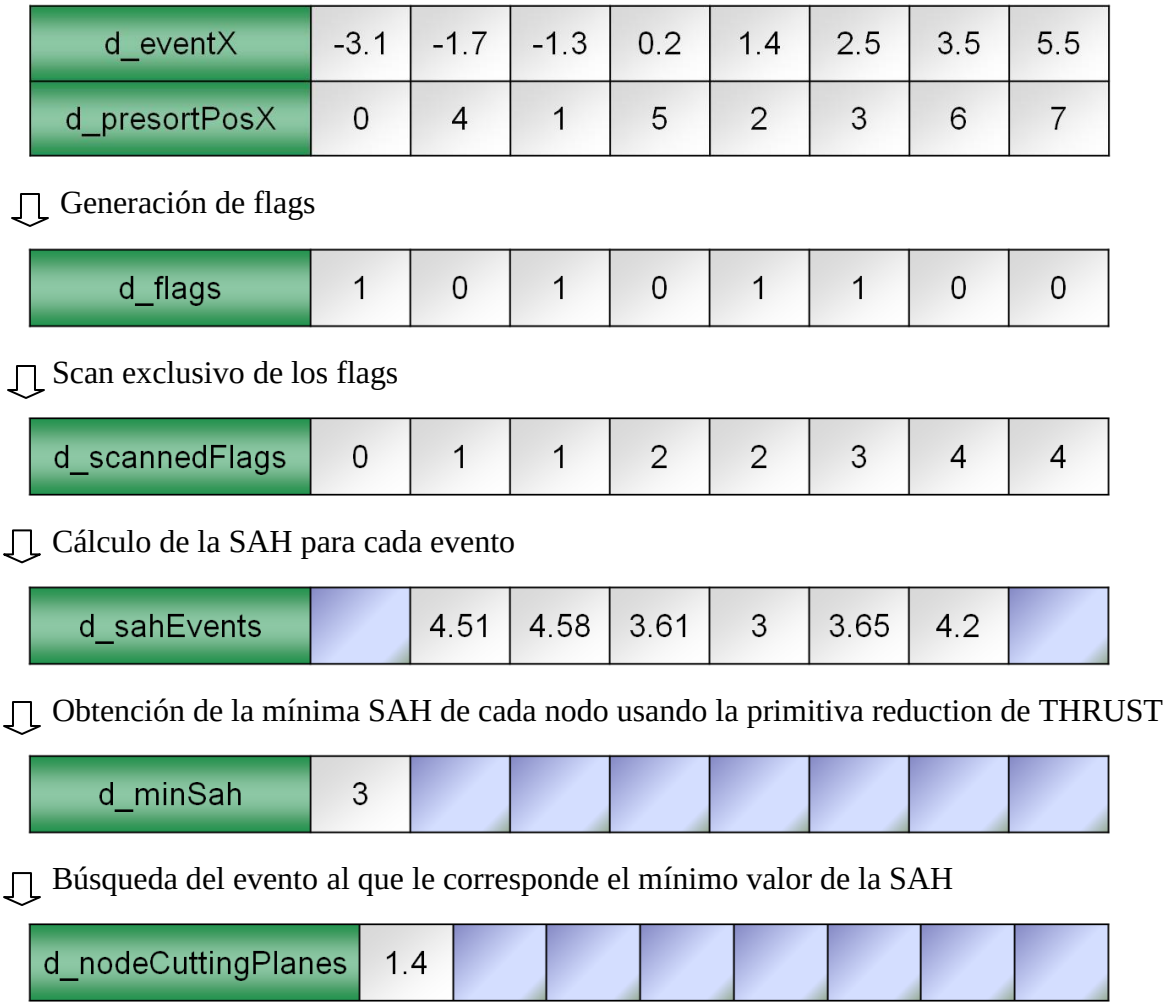
**Algoritmo 3.6 Kernel para el cálculo de SAH de cada evento.**

```
00  __global__ void KernelCalculateSAH(
01      float*          d_sahEvents,
02      unsigned int*   d_scannedFlags,
03      float*          d_eventX,
04      unsigned int*   d_nodeForEvent,
05      unsigned int*   d_nodeInitialPos,
06      unsigned int*   d_nodeNumSegments,
07      unsigned int    numEvents)
08  {
09      // Número de hilo
10      unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
11      if (i < numEvents) {
12          // Se mira el nº de nodo del evento y si es distinto de UNUSED, se realiza tarea
13          unsigned int nodeNumber = d_nodeForEvent[i];
14          if (nodeNumber != UNUSED) {
15              float sahValue; // Registro para el valor de sah
16              // Tomamos la posición del nº de segmentos del nodo en cuestión
17              // si tiene pocos segmentos (valor umbral) su valor sah es alto: UNUSED
18              unsigned int numSegments = d_nodeNumSegments[nodeNumber];
19              unsigned int initialEventPos = d_nodeInitialPos[nodeNumber];
20              if ((numSegments <= 1) || (initialEventPos == UNUSED))
21                  sahValue = UNUSED;
22              else {
23                  float eventX = d_eventX[i]; // Valor del evento
24                  float initialEvent = d_eventX[initialEventPos];
25                  float lastEvent = d_eventX[initialEventPos + (numSegments << 1) - 1];
26                  // Número de eventos START a la izquierda de este evento para este nodo
27                  unsigned int scannedFlag = d_scannedFlags[i] - d_scannedFlags[initialEventPos];
28                  // Si el evento coincide en valor con el último o el primero su SAH es UNUSED
29                  // porque no valen como candidatos los bordes de la caja padre
30                  if ( (lastEvent == eventX) || (initialEvent == eventX) )
31                      sahValue = UNUSED;
32                  else {
33                      // Si no, se calcula su valor de SAH:
34                      //     Sah(event) = 1 + ( (NL * AABBL + NR * AABBR) / AABBFather )
35                      sahValue = SAH(
36                          (float) scannedFlag, // NL
37                          eventX - initialEvent, // AABBL
38                          (float)(numSegments - (i - initialEventPos - scannedFlag)), // NR
39                          lastEvent - eventX, // AABBR
40                          lastEvent - initialEvent); // AABBFather
41                      // Se mira si cumple el umbral de SAH
42                      if (sahValue > numSegments)
43                          sahValue = UNUSED;
44                  }
45              }
46
47              // Escritura a la memoria global al final para hacerla
48              // de forma fusionada por warp
49              d_sahEvents[i] = sahValue;
50          }
51      }
52  }
```

### *Obtención de las mínimas SAH*

Siguiendo el ejemplo de la Figura 3.5, en "d_sahEvents" están los resultados de la SAH para cada evento de d_eventX, marcando en negro los valores UNUSED (los bordes de la caja del padre). Ese array con valores SAH es pasado a la primitiva de "reduction" de THRUST para hallar el mínimo valor. Después se busca a qué evento corresponde, en este caso al evento con valor 1,4. Ése es el valor que se toma como punto de corte para el nodo.

**Figura 3.5 Ejemplo de la obtención del punto de corte para el nodo raíz con los segmentos de la Figura 3.1.**



## División de nodos

### *Generación de flags*

Con el punto de corte de cada nodo se averigua a qué hijo va a ir cada segmento: al izquierdo, al derecho o a ambos, si el punto de corte está dentro del segmento. En este último caso, el segmento se duplica, yendo cada copia al hijo correspondiente.

El kernel del Algoritmo 3.7 muestra la manera de averiguar a qué lado va cada segmento. Primero se comprueba si se eligió un punto de corte (línea 22). Si no se eligió un punto de corte se trata de un nodo hoja. En esta versión, por decisión propia de diseño, los nodos hojas mantienen sus eventos en el array de eventos durante todo el algoritmo. Los eventos de los nodos hoja se marcan como pertenecientes al lado izquierdo (líneas 25 y 26), y así, reservan el espacio que necesitan para copiarse en la estructura de entrada de la siguiente iteración. En la versión 2D/3D se copian en otra estructura para la solución final. Si no se trata de una hoja, solo los eventos end comprueban a qué lado van a ir tanto su pareja start como él. La posición de su pareja start se obtiene del array d_pairEvent (línea 31). Con el evento start y end se hace la comprobación (líneas 32-37), cuyo resultado se escribe en los arrays de flags izquierda y derecha (líneas 39-42).

Con este algoritmo sólo trabajan los eventos end. Esto es poco eficiente porque los hilos lanzados para los eventos start no hacen nada y se desaprovecha recursos de la GPU. Este enfoque se mejora en las versiones de dos y tres dimensiones, lanzando hilos por cada triángulo en vez de por cada evento. En el kernel, para evitar que la bifurcación sea larga se inicializan los valores a escribir en los array de flags a 1, de forma que sólo es necesario determinar los casos en los que debemos ponerlos a 0. Como se pudo ver en el capítulo de GPU los hilos de un warp ejecutan las mismas instrucciones, pasando por ambos caminos de la bifurcación, y de ahí la importancia de acortar el número de instrucciones dentro de una sección condicional.

Para describir cómo dividimos los nodos vivos usando los arrays d_leftFlags y d_rightFlags, avanzamos en el ejemplo que nos ocupa hasta la segunda iteración. En la Figura 3.6 se muestra el array con eventos para el nodo 1 y el nodo 2 (hijos izquierdo y derecho de la raíz) y los puntos de corte elegidos, tanto el que se eligió para el nodo raíz en la anterior iteración (en verde) y que sirvió para generar los nodos 1 y 2, como los puntos de corte elegidos para estos nodos en esta iteración (en violeta). En la posición 1 de “d_nodeInitialPos” se marca el comienzo

del nodo 1 en el array de eventos (0) y en la posición 2 el del nodo 2 (4). Tras la generación de flags se realiza el scan exclusivo cuyo resultado se guarda en d_leftScannedFlags y d_rightScannedFlags.

**Algoritmo 3.7 Kernel para la generación de flags sobre a qué lado va cada evento.**

```
00    __global__ void KernelGenFlags(
01          unsigned int*   d_leftFlags,
02          unsigned int*   d_rightFlags,
03          float*            d_eventX,
04          unsigned int*   d_eventXOwner,
05          float*            d_bestPlanes,
06          unsigned int*   d_nodeForEvent,
07          unsigned int*   d_pairEvent,
08          unsigned int    numEvents)
09    {
10          // Número de hilo
11          unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
12          // Solo para el rango de eventos a tratar
13          if (i < numEvents)  {
14                unsigned int nodeNumber = d_nodeForEvent[i];
15                if (d_nodeForEvent[i] != UNUSED) {
16                      // Se guarda el punto de corte en un registro y se comprueba si pertenece
17                      // a un nodo que va a hacer split. Si es UNUSED no hace split
18                      float plane = d_bestPlanes[nodeNumber];
19                      // También se inicializan los valores de los flags a 1 y 1 para tener menos
20                      // if anidados y usar menos instrucciones
21                      unsigned int leftFlag = 1, rightFlag = 1;
22                      if (plane == UNUSED) {
23                            // Los nodos hojas de anteriores iteraciones se dejan
24                            // como están (marcando el lado izquierdo)
25                            d_leftFlags[i] = 1;
26                            d_rightFlags[i] = 0;
27                      }
28                      else {
29                            // Se mira si su posición en d_AABB es mayor que numSegments (evento END)
30                            if (d_eventXOwner[i] > numSegments) {
31                                  unsigned int pairEvent = d_pairEvent[i];
32                                  if (d_eventX[i] <= plane)       // Va al lado izquierdo, el eje está a
33                                                                  //su derecha  Start----End  |
34                                        rightFlag = 0; // Se marca que a la derecha no está
35                                  // Puede ser un duplicado, es necesario mirar su start
36                                  else if (plane  <= d_eventX[pairEvent])   // |   Start----End
37                                              leftFlag = 0; // Se marca que a la izquierda no está
38                                        // si no: Start--|--End
39                                  d_leftFlags[i] = leftFlag;
40                                  d_rightFlags[i] = rightFlag;
41                                  d_leftFlags[pairEvent] = leftFlag;
42                                  d_rightFlags[pairEvent] = rightFlag;
43                            }
44                      }
45                }
46          }
47    }
```

### *Copia en la estructura de swap*

Por último, hay que copiar los eventos en otra estructura para usarlos en la siguiente generación. Es necesario emplear otra estructura que sirva de swap porque si cada hilo modifica valores del propio array se producen inconsistencias. Esto sólo es necesario para los arrays asociados a eventos, ya que los arrays asociados a nodos se mantienen intactos sin problema.

La copia necesita saber la posición que le toca a cada evento de la siguiente generación. Se emplean los arrays d_leftScannedFlags y d_rightScannedFlags en los que la posición de cada evento se determina según el hijo al que pertenece:

- Si pertenece sólo al hijo izquierdo, su posición es el valor en d_leftScannedFlags.
- Si pertenece sólo al hijo derecho, como los nodos derechos se escriben a continuación de las posiciones de los nodos izquierdos, su posición es el valor en d_rightScannedFlags más el número total de eventos que van a la izquierda (último valor en d_leftScannedFlags más el último valor en d_leftFlags para completar el scan exclusivo).
- Si es un evento duplicado, hay que generar las copias para los dos hijos. Para el hijo izquierdo, el evento start no se altera, pero el evento end es el punto de corte. Para el hijo derecho, al contrario, el evento end no se modifica y el evento start es el punto de corte.
- Si pertenece a un nodo hoja, su posición es el valor en d_leftScannedFlags.

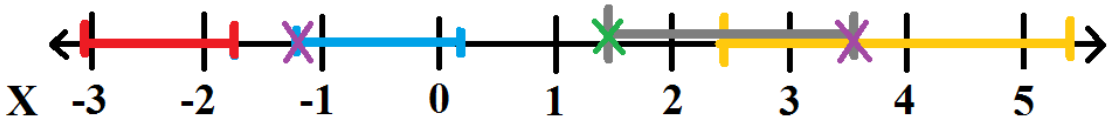
En la Figura 3.6 el segmento naranja es cortado por el punto de corte, por lo que es un segmento duplicado. El hijo izquierdo continuará la siguiente iteración del algoritmo con los eventos que delimitan la parte izquierda del punto de corte (2.5 y 3.5), mientras que el hijo derecho con los eventos de la parte derecha (3.5 y 5.5). Como ejemplo, para el evento 5.5 de tipo end (en la posición 7 de d_eventX(in)) hay que obtener la posición en d_eventX (out) para el evento end generado para el hijo izquierdo (3.5) y para el evento end generado para el hijo derecho (5.5). Para el evento del hijo izquierdo (3.5), es suficiente con el valor de su posición en el array d_leftScannedFlags (5). Por lo tanto, se escribe en la posición 5 de d_eventX (out) el valor 3.5 y el nodo asociado a ese evento en d_nodeForEvent (out), (2 * 2 + 1 = 5), ya que 2 es el nodo que se divide. Para el evento del hijo derecho (5.5), no es suficiente con el valor en el array d_rightScannedFlags (3). Los eventos de los nodos hijos derechos se escriben a continuación de los eventos de los hijos izquierdos, es decir, para saber la posición final de un evento de un hijo

derecho es necesario sumar las posiciones ocupadas por los eventos generados para todos los hijos izquierdos de la próxima iteración (6 posiciones), por lo tanto la posición final del evento 5.5 es (6 + 3 = 9), que es la posición que tiene en la Figura 3.4 en el array d_eventX (Out) y cuyo nodo es el hijo derecho del nodo 2 (2 * 2 + 2 = 6).

Además de copiar los eventos en la estructura de entrada de la siguiente iteración, también es necesario copiar la posición que tenían antes de ordenarse al comienzo del algoritmo. Es importante mantener esta información en d_presortPosX (out) porque sirve para determinar a qué segmento pertenece y si se trata de un evento start o end, tal y como he explicado en la sección de "Inicialización". Por ejemplo, el evento en el índice 0 con valor 0 es un evento start y pertenece al segmento número 0 (el rojo), y el evento en el índice 1, con valor 4 es un evento end y también pertenece al segmento número 0.

Finalmente se establecen las AABBs de los nuevos nodos. Para cada nodo se emplea el menor y mayor evento para obtener la caja mínima que lo envuelve. De este modo, se consigue eliminar regiones vacías de los nodos del árbol, de forma que si un rayo cae en una región vacía no comprueba intersección con segmentos. Estas regiones de espacio vacío se pueden generar con los puntos de corte. Por ejemplo en la Figura 3.6, el punto de corte del nodo raíz (verde) genera un espacio vacío entre 0.2 y 1.4.

**Figura 3.6 Ejemplo para una segunda iteración del algoritmo. La marca en verde es el punto de corte del nodo raíz en la primera iteración. Las marcas en violeta son los puntos de corte elegidos para los hijos izquierdo y derecho del raíz en la segunda iteración.**




| d_eventX (in) | -3.1 | -1.7 | -1.3 | 0.2 | 1.4 | 2.5 | 3.5 | 5.5 |
|---|---|---|---|---|---|---|---|---|
| d_nodeForEvent (in) | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeInitialPos | | 0 | 4 | | | | |
| d_nodeCuttingPlanes | 1.4 | -1.3 | 3.5 | | | | |

⇩ Generación de flags

| d_leftFlags | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| d_rightFlags | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

⇩ Scan exclusivo de los flags

| d_leftScannedFlags | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| d_rightScannedFlags | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 3 |

⇩ Copia de los eventos y actualización de estructuras para los nuevos nodos

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeInitialPos | | | | 0 | 6 | 2 | 8 |
| d_nodeNumSegments | | | | 1 | 1 | 2 | 1 |

| d_nodeForEvent (out) | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 4 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| d_eventX (out) | -3.1 | -1.7 | 1.4 | 2.5 | 3.5 | 3.5 | -1.3 | 0.2 | 3.5 | 5.5 |
| d_presortPosX (out) | 0 | 4 | 2 | 3 | 7 | 6 | 1 | 5 | 3 | 7 |
| d_numEvents | 10 | | | | | | | | | |

⇩ Se establecen las AABB de los nuevos nodos.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeAABBXMin | -3.1 | -3.1 | 1.4 | -3.1 | -1.3 | 1.4 | 3.5 |
| d_nodeAABBXMax | 5.5 | 0.2 | 5.5 | -1.7 | 0.2 | 3.5 | 5.5 |

## Finalización

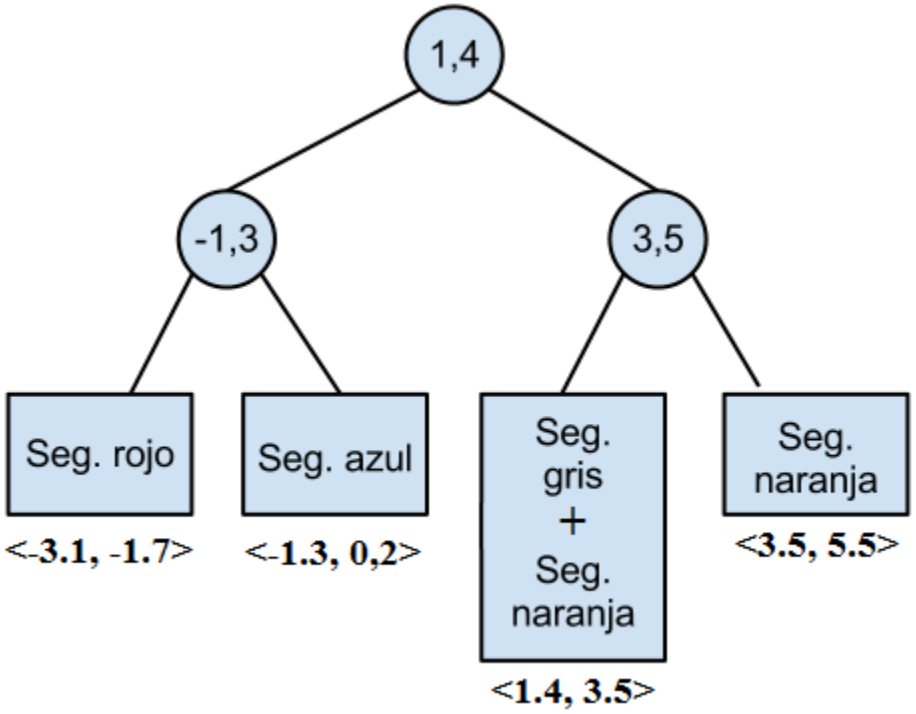
Tras salir del bucle del algoritmo principal se obtienen, copiando al host, las estructuras necesarias a partir de las cuales interpretar un kd-tree. Estas son:

- d_nodeInitialPos: para saber el comienzo de cada nodo hoja en el array d_eventX.
- d_nodeCuttingPlanes: se emplea para navegar por cada nodo del árbol. Si el nodo tiene marca UNUSED se trata de un nodo hoja, y si no, se trata del punto de corte del nodo interno.
- d_nodeNumSegments: para saber el número de segmentos que contiene cada nodo del árbol.
- d_nodeAABBXMin y d_nodeAABBXMax: determinan la caja mínima que envuelve al nodo.
- d_eventX: contiene los eventos de los nodos hoja.
- d_presortPosX: este array contiene la posición que tenía el evento al comenzar el algoritmo después de calcular las AABBs de los segmentos y antes de ordenar el array d_event, lo que determina el segmento al que pertenece. Por ejemplo, el evento en la posición 1, pertenece al segmento 1. De este modo se identifica el segmento que le corresponde a cada evento de las hojas.

La Figura 3.7 muestra el árbol resultante del ejemplo de la Figura 3.6 y las estructuras que han resultado tras la ejecución del algoritmo de construcción. En (a), en los nodos internos aparecen los puntos de corte. En las hojas los segmentos a los que hacen referencia y la región que abarca el nodo. Las regiones que no están contenidas en los nodos hoja pertenecen a espacio vacío y son resultado de la técnica de "Maximización de espacio vacío".

En la tabla (b), el array d_nodeInitialPos contiene las posiciones de comienzo en d_eventX de los eventos de las hojas, y en d_nodeNumSegments el número de segmentos que contiene cada una. En d_nodeCuttingPlanes se guardan los puntos de corte de los nodos internos, en este caso el del nodo raíz (0) y sus hijos izquierdo (1) y derecho (2). En d_nodeAABBXMin/Max se guardan los límites de las AABBs de cada nodo. Por último, para conocer a qué segmento pertenece cada evento de las hojas, se emplea el array d_presortPosX. Este array contiene las posiciones que tenían los eventos al comienzo del algoritmo, lo que determina el número de segmento al que pertenecían. El array no forma parte de la solución, pero se incluye en la figura para facilitar su comprensión y contiene el nodo al que pertenece cada evento.

**Figura 3.7 a) Árbol generado para el ejemplo de la Figura 3.6 y b) las estructuras resultantes tras la ejecución del algoritmo.**




(a)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeInitialPos | | | | 0 | 6 | 2 | 8 |
| d_nodeNumSegments | | | | 1 | 1 | 2 | 1 |
| d_nodeCuttingPlanes | 1.4 | -1.3 | 3.5 | | | | |
| d_nodeAABBXMin | -3.1 | -3.1 | 1.4 | -3.1 | -1.3 | 1.4 | 3.5 |
| d_nodeAABBXMax | 5.5 | 0.2 | 5.5 | -1.7 | 0.2 | 3.5 | 5.5 |

| d_nodeForEvent | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 4 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| d_presortPosX | 0 | 4 | 2 | 3 | 7 | 6 | 1 | 5 | 3 | 7 |
| d_eventX | -3.1 | -1.7 | 1.4 | 2.5 | 3.5 | 3.5 | -1.3 | 0.2 | 3.5 | 5.5 |

(b)

# Capítulo 4 - Construcción de Kd-trees para triángulos en 2D

Respecto a la versión 1D, la versión 2D amplía el tamaño de las estructuras que almacenan los puntos y los eventos, conteniendo, además de las X, las componentes Y. Aunque los algoritmos tienen que distinguir sobre qué componente actuar, la tarea a realizar sobre cada eje es la misma. Las versiones 2D y 3D son muy similares entre sí. Para la versión 3D, se amplían las estructuras con la componente Z, y los algoritmos hacen una tercera distinción (para el eje Z).

Para 2D se emplean triángulos (también denominados *primitivas*) en vez de segmentos, y el espacio se parte mediante rectas en vez de puntos. A continuación se explican los detalles del caso 2D.

## Descripción del algoritmo

El primer cambio del algoritmo (ver Algoritmo 4.1) es que ahora se tiene en cuenta otra condición de parada. Si el algoritmo se queda sin eventos para los cuales realizar trabajo (línea 5), se termina aunque no se haya alcanzado el número de iteraciones predicho. La versión 1D no se quedaba sin eventos, porque los nodos hojas se mantenían en el array de eventos aunque no realizaran trabajo útil. En esta versión, se trasladan los nodos hoja a otra estructura que forma parte de la solución final, quitándolos del array de eventos. Se puede considerar entonces que este enfoque mejora el algoritmo propuesto para el caso 1D.

El segundo cambio es que la división de nodos se realiza en dos fases (líneas 10 y 16). Como se explicará en la sección “División de nodos”, las rectas de corte producen desorden en los arrays de eventos. La primera fase inicia la ordenación y hace la copia a la estructura de entrada de la siguiente iteración. La segunda fase termina de ordenarlos.

Para terminar, si el algoritmo alcanza la cota superior estimada del número de niveles a generar (explicada en la sección “Pasos previos” para el caso 1D) y tiene nodos que pueden seguir dividiéndose (línea 19), es necesario convertirlos en nodos hoja, trasladándolos a la estructura que los almacena. Como se explicó para el caso 1D, cuando existe una concentración alta de triángulos se produce un mayor número de duplicados, necesitando más niveles para completar la construcción mediante el uso de SAH que los que la estimación calcula.

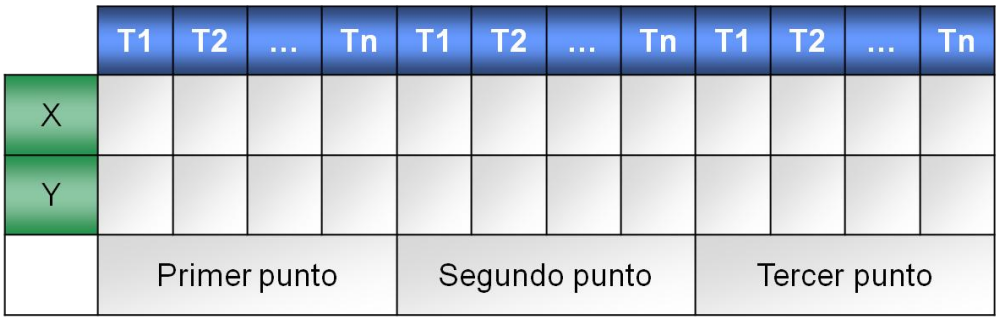
**Algoritmo 4.1 Código principal para 2D y 3D.**

```
00    KDTree* KDTreeGenerator::genTree()
01    {
02          // Inicialización de estructuras
03          init();
04          // Creación del árbol por niveles: mientras haya nodos vivos y no se haya superado la cota
05          while ((memPredictor->getNumEvents() > 0)
06                && (memPredictor->getActualLevel() < memPredictor->getMaxLevels() - 1)) {
07                // Selección de los planos de corte para cada nodo del nivel
08                selectBestPlane();
09                // Primera fase de la división de nodos
10                splitFirstPhase();
11                // Avanza los iteradores
12                memPredictor->nextLevel(d_numEvents);
13                // Intercambia los arrays de swap sobre los que escribir
14                swap();
15                //  Segunda fase de la división de nodos, reordenación de eventos de la siguiente iteración
16                splitSecondPhase();
17          }
18          // Si la cota del número de niveles no es suficiente y hay nodos vivos, se les convierte en hoja
19          if (memPredictor->getNumEvents() > 0) {
20                finishAliveNodes();
21          }
22          // Extrae del device el árbol resultante
23          return end();
24    }
```

## Pasos previos

Como paso previo al Algoritmo 4.1, es necesario copiar los puntos de los triángulos a la memoria de la GPU (con cudaMemCpy). La Figura 4.1 muestra la estructura que almacena los puntos en un array para el eje X y otro para el eje Y. Esta estructura favorece las lecturas fusionadas por warp. Cuando los 32 hilos de un warp acceden al primer punto de su primitiva, al estar en memoria contigua, sólo se realiza una transacción de memoria para los 32 datos.

**Figura 4.1 Arrays para almacenar los puntos de cada triángulo.**



La reserva de memoria emplea las mismas cotas superiores que para 1D. El único cambio es que la componente Y duplica estructuras, por ejemplo, ahora tendremos d_eventX y d_eventY.

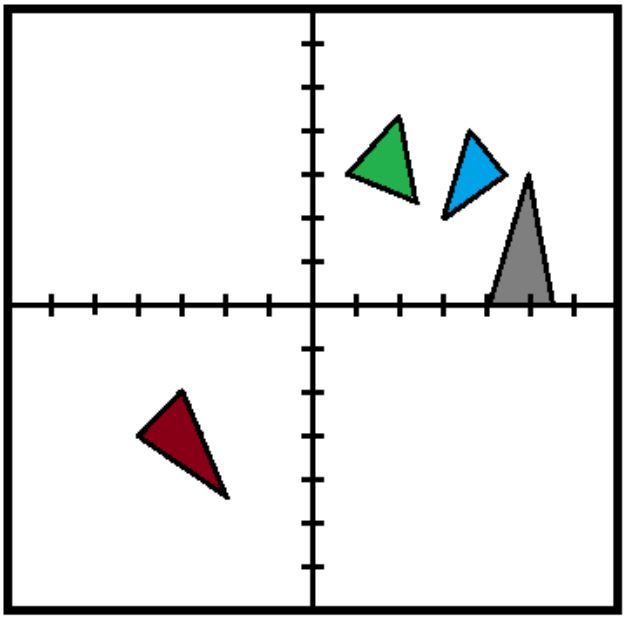
## Inicialización

La inicialización sigue los mismos pasos que la versión 1D: requiere calcular las AABBs de los triángulos, ordenar las listas de eventos e inicializar estructuras.

### *Cálculo de AABBs*

Cada triángulo tiene una caja determinada por cuatro eventos, dos para los límites en el eje X (xStart, xEnd), y dos en el eje Y (yStart, yEnd). Se procede de forma parecida a 1D pero teniendo que hacer más cálculos ya que cada primitiva tiene tres puntos (el segmento tiene dos), y hay dos coordenadas. La Figura 4.2 muestra un ejemplo con cuatro triángulos en 2D, el contenido de las estructuras que almacenan los puntos desde la entrada y el resultado de calcular las AABBs. En la primera tabla se muestra la estructura que almacena los puntos. Para el triángulo verde (T2), su primer punto es (2, 4.5), el segundo (0.7, 3) y el tercero (2.2, 2.5). Por ejemplo, de los eventos señalados en rojo para el eje X, se escriben en d_AABBX el menor (0.7) en la parte de eventos start y el mayor (2.2) en la de end.

**Figura 4.2 Ejemplo de triángulos en dos dimensiones, el contenido de estructuras que almacenan los puntos y el resultado de sus AABBs.**



| | T1 | T2 | T3 | T4 | T1 | T2 | T3 | T4 | T1 | T2 | T3 | T4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | -4 | 2 | 3.5 | 4 | -3 | 0.7 | 3 | 5 | -2 | 2.2 | 4.5 | 5.5 |
| Y | -3.5 | 4.5 | 4 | 0 | -2 | 3 | 2 | 3 | -4,5 | 2.5 | 3 | 0 |
| | Primer punto | | | | Segundo punto | | | | Tercer punto | | | |

⇩ Cálculo de las AABBs

| | T1 | T2 | T3 | T4 | T1 | T2 | T3 | T4 |
|---|---|---|---|---|---|---|---|---|
| índices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| d_AABBX | -4 | 0.7 | 3 | 4 | -2 | 2.2 | 4.5 | 5.5 |
| d_AABBY | -4.5 | 2.5 | 2 | 0 | -2 | 4.5 | 4 | 3 |
| | Eventos start | | | | Eventos end | | | |

⇩ Ordenación de eventos

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| d_eventX | -4 | -2 | 0.7 | 2.2 | 3 | 4 | 4.5 | 5.5 |
| d_presortPosX | 0 | 4 | 1 | 5 | 2 | 3 | 6 | 7 |
| d_eventY | -4.5 | -2 | 0 | 2 | 2.5 | 3 | 4 | 4.5 |
| d_presortPosY | 0 | 4 | 3 | 2 | 1 | 7 | 6 | 5 |

El Algoritmo 4.2 muestra el kernel encargado de realizar el cálculo de las AABBs. El kernel es lanzado con 2 * numTriangles hilos para realizar mayor trabajo en paralelo. Cada triángulo tiene 2 hilos, uno para calcular los límites en el eje X y otro en el eje Y. Los primeros numTriangles (líneas 14-17) inicializan los punteros para el eje X, los segundos numTriangles hilos los inicializan para el eje Y (líneas 20-22). Después se calculan las posiciones de los tres puntos (líneas 24-25, la posición del primer punto coincide con el índice del triángulo), se calculan el mínimo y el máximo y se guardan (27-35).

**Algoritmo 4.2 Código para calcular las AABBs que distribuye el trabajo para el eje X y el eje Y en paralelo.**

```
00 __global__ void KernelComputeAABB(
01    float*              d_AABBX,
02    float*              d_AABBY,
03    float*              d_pointsX,
04    float*              d_pointsY,
05    unsigned int    numTriangles)
06 {
07    unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
08
09    if (i < (2 * numTriangles)) {
10
11         float* d_AABB, * d_points;
12         unsigned int indexTriangle;
13
14           if (i < numTriangles) {
15                 d_AABB = d_AABBX;
16                 d_points = d_pointsX;
17                 indexTriangle = i;
18           }
19           else {
20                 d_AABB = d_AABBY;
21                 d_points = d_pointsY;
22                 indexTriangle = i - numTriangles;
23           }
24           unsigned int secondPointPos = indexTriangle + numTriangles;
25           unsigned int thirdPointPos = secondPointPos + numTriangles;
26
27           float first = d_points[indexTriangle];
28           float second = d_points[secondPointPos];
29           float third = d_points[thirdPointPos];
30
31           float min, max;
32          ... {cálculo de mínimo y máximo}...
33
34           d_AABB[indexTriangle] = min;
35           d_AABB[secondPointPos] = max;
36     }
37 }
```

### *Ordenación de eventos*

El siguiente paso consiste en obtener las listas ordenadas de eventos. En 1D se obtiene la lista ordenada para eventos solo en el eje X (d_eventX). En 2D, además de esa, se obtiene la lista ordenada en el eje Y (d_eventY). El procedimiento es el mismo, repitiéndolo para el eje Y, guardando la posición que tenía cada evento antes de ordenar (d_presortPos), tal y como muestra

la Figura 4.2. Por ejemplo, para el triángulo gris (T4), su evento xStart (4) con índice 3, una vez ordenado en d_eventX está en la posición 5 y tiene en d_presortPosX un valor de 3 (el índice previo).

### *Inicialización de otras estructuras*

En cuanto a la inicialización de otras estructuras para los nodos (ver Figura 4.3), se mantienen las mismas que en 1D, duplicando para las AABBs con el eje Y:

- En d_nodeNumPrimitives se guarda el número de triángulos que contiene el nodo.
- En d_nodeAABBXMin/XMax/YMin/YMax, se guardan las AABBs de cada nodo. Los valores se obtienen con el primer y último evento del nodo en cada eje.
- En d_nodeInitialPos, si se trata de un nodo vivo, se guarda la posición de comienzo en el array de eventos. Si el nodo es hoja, indica la posición de comienzo en el array de hojas.

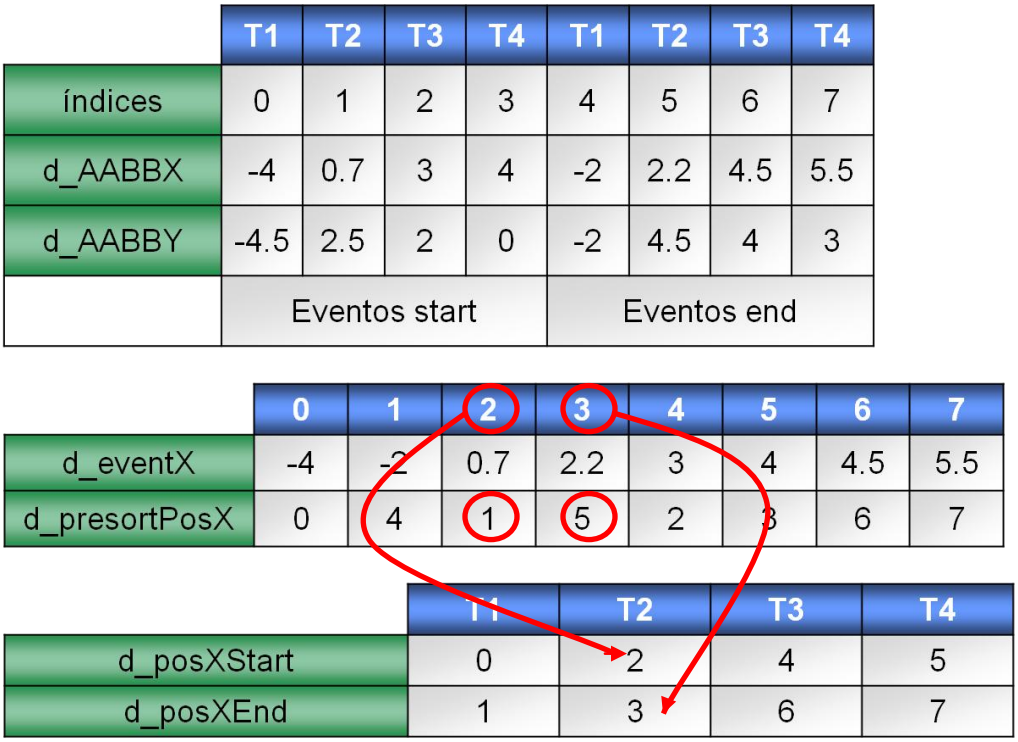
**Figura 4.3 Estructuras para nodos inicializadas según el ejemplo de la Figura 4.2.**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeNumPrimitives | 4 | | | | | | |
| d_nodeAABBXMin | -4 | | | | | | |
| d_nodeAABBXMax | 5.5 | | | | | | |
| d_nodeAABBYMin | -4.5 | | | | | | |
| d_nodeAABBYMax | 4.5 | | | | | | |
| d_nodeInitialPos | 0 | | | | | | |

Para las estructuras para los eventos, en esta versión he seguido un enfoque que difiere de la versión 1D. En vez de estar asociadas a cada evento, las estructuras se asocian a cada primitiva. Cada primitiva tendrá así acceso a la posición que ocupan sus cuatro eventos (d_posXStart,d_posXEnd, d_posYStart, d_posYEnd) en el array de eventos, y al número de triángulo al que hacen referencia (d_primitiveNumber). Para calcular estas posiciones se usa el kernel del Algoritmo 4.3 que lanza un hilo por cada evento, es decir, 2 x numEvents hilos, donde

los primeros numEvents hilos se encargan del eje X, y los siguientes del eje Y. La Figura 4.4 muestra un ejemplo de cómo se calculan estas posiciones a partir de los arrays d_eventX y d_presortPosX. Con d_presortPosX, cada evento conoce la posición que tenía antes de ordenarse. Con esa posición determina a qué primitiva pertenecía y si era un evento start o end. Por ejemplo, el evento de la posición 2 de d_eventX (0.7) estaba en la posición 1 antes de la ordenación, lo que quiere decir que su triángulo era T2. Ya que 1 es menor que el número de triángulos, pertenece a la parte de eventos start. Para el evento de la posición 3 de d_eventX (2.2) que tenía posición 5, su triángulo es T2 y se trata de un evento end.

**Figura 4.4 Ejemplo de obtención de las posiciones de los eventos en el eje X. La primera tabla contiene las AABBs de cada triángulo. La segunda los eventos en el eje X ordenados (d_eventX) y la posición que tenían antes de la ordenación (d_presortPosX). La tercera tabla contiene las posiciones de los eventos para cada primitiva.**



**Algoritmo 4.3 Kernel que obtiene las posiciones de los eventos de cada primitiva.**

```
00 __global__ void KernelInitEventPositions(
01    unsigned int*   d_posXStart,   unsigned int*   d_posXEnd,
02    unsigned int*   d_posYStart,   unsigned int*   d_posYEnd,
03    unsigned int*   d_preorderPosX,       unsigned int*   d_preorderPosY,
04    unsigned int    numPrimitives, unsigned int    numEvents)
05{
06    unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
07    if (i < (2 * numEvents)) {
08
09          unsigned int* d_posStart;
10          unsigned int* d_posEnd;
11          unsigned int* d_preorderPos;
12          unsigned int indexEvent;
13
14          if (i < numEvents) {
15                d_posStart = d_posXStart;
16                d_posEnd = d_posXEnd;
17                d_preorderPos = d_preorderPosX;
18                indexEvent = i;
19          }
20          else {
21                d_posStart = d_posYStart;
22                d_posEnd = d_posYEnd;
23                d_preorderPos = d_preorderPosY;
24                indexEvent = i - numEvents;
25          }
26
27          unsigned int primitiveNumber = d_preorderPos[indexEvent];
28          if (primitiveNumber < numPrimitives) {
29                d_posStart[primitiveNumber] = indexEvent;
30          }
31          else {
32                primitiveNumber -= numPrimitives;
33                d_posEnd[primitiveNumber] = indexEvent;
34          }
35    }
36}
```

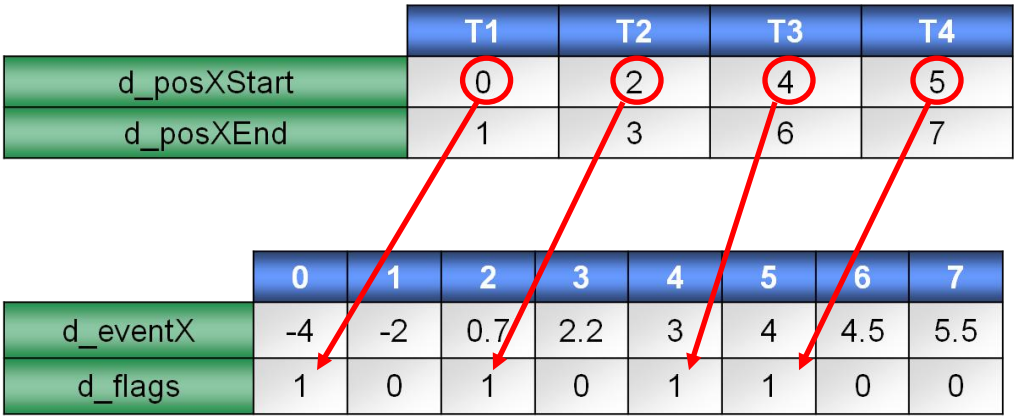
## Selección de mejores rectas de corte

Con dos dimensiones todos los eventos en los ejes X e Y son candidatos a rectas de corte. Por ejemplo, un evento en el eje Y con valor 0.7 equivale a la recta Y = 0.7 para el corte. Los pasos son los mismos que en 1D: generación de flags, scan exclusivo sobre flags, cálculo de SAH y obtención de los eventos cuyos valores de SAH son mínimos para cada nodo.

### *Generación de flags y scan exclusivo*

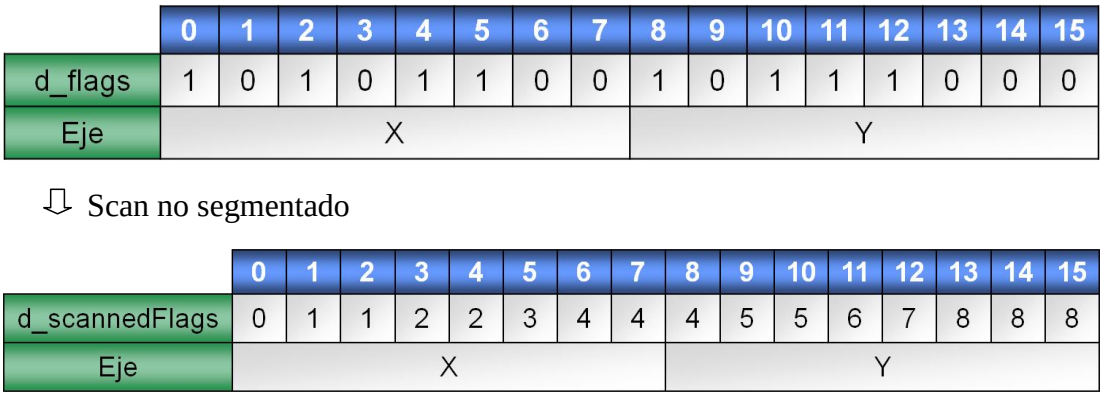
En 1D, la generación de flags se hacía por evento. En esta versión cada primitiva tiene dos hilos, uno para escribir los flags de los eventos en el eje X (xStart, xEnd) y otro para los del eje Y (yStart, yEnd). La Figura 4.5 muestra un ejemplo para el eje X en el que se utiliza la posición de los eventos de cada primitiva para completar el array de flags (1 para eventos start y 0 para eventos end).

**Figura 4.5 Ejemplo de generación de flags para el eje X.**




El array de flags reúne en memoria contigua (en un mismo array) los valores para todos los ejes. La razón es que es más eficiente hacer un único scan sobre 2 * numEvents elementos que hacer dos scans (uno para X y otro para Y) sobre numEvents datos. La Figura 4.6 muestra la estructura de estos arrays para flags de 8 eventos en X y 8 en Y al aplicar la función de scan exclusivo de THRUST. Sin embargo surge un problema. El objetivo es obtener el número de primitivas que hay a cada lado de cada evento, y los flags de Y no tienen nada que ver con los flags de X. Interesaría que la cuenta comenzara desde cero para la parte del eje Y. La solución consiste en que en cada acceso a la parte del eje Y hay que restar el valor de la primera posición en este eje. Por ejemplo, en la Figura 4.6 (d_scannedFlags), para conocer el número de primitivas del quinto evento en el eje Y (posición 12) hay que restar a 7 el valor de la primera posición, en el eje Y (posición 8), es decir 4. Por lo que 3 es el número de primitivas que hay al lado izquierdo de ese evento, en el eje Y.

**Figura 4.6 Ejemplo de generación de flags para el eje X.**




### *Cálculo de las SAH*

A continuación se calculan los valores de la SAH para cada evento. Para esta tarea se lanza un solo hilo por cada primitiva. Ese hilo calcula la SAH para los cuatro eventos de la primitiva (xStart, xEnd, yStart, yEnd) de manera secuencial (ver Algoritmo 4.4, líneas 34-49). He seguido este enfoque, frente al mayor grado de paralelización que supone calcular en paralelo la SAH de los 4 eventos, porque cada primitiva va a elegir como candidato al mejor de sus cuatro eventos. Con ello se consigue reducir cuatro veces el tamaño del array de salida, y por lo tanto, al aplicar la operación reduction de THRUST para hallar los mínimos, el algoritmo es mucho más rápido.

Durante el desarrollo de este kernel, hice otra versión que lanzaba 4 hilos por primitiva. Cada hilo calculaba la SAH de uno de los eventos de la primitiva. Tras ello, y usando una operación de sincronización de hilos, el primer hilo asociado a la primitiva calculaba el candidato de entre los cuatro eventos. El coste de sincronizar los hilos hacía que el tiempo de ejecución fuera de casi el doble que la versión que se presenta en el Algoritmo 4.4.

Para el cálculo de coste de la SAH, como se mencionó antes, el acceso a la parte del eje Y para el array d_scannedFlag requiere tener en cuenta el desplazamiento que se produce sobre la parte del array del eje X. En el Algoritmo 4.5, se usa el parámetro offset para corregir este desplazamiento (líneas 13-14). El parámetro vale 0 para el eje X, y numEvents para el eje Y.

**Algoritmo 4.4 Código para el cálculo de SAH.**

```
00 __global__ void KernelCalculateSAH(
01    key_value*       d_sahEvents,          unsigned int*   d_scannedFlags,
02    unsigned int*    d_primitiveNode,      unsigned int*   d_posXStart,
03    unsigned int*    d_posXEnd,            unsigned int*   d_posYStart,
04    unsigned int*    d_posYEnd,            float*          d_eventX,
05    float*           d_eventY,             unsigned int*   d_nodeInitialPos,
06    unsigned int*    d_nodeNumTriangles, unsigned int      numPrimitives,
07    unsigned int     numEvents)
08 {
09    unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
10
11    if (i < numPrimitives) {
12          unsigned int nodeNumber = d_primitiveNode[i];
13          unsigned int nodeNumTriangles = d_nodeNumTriangles[nodeNumber];
14
15          if (nodeNumTriangles <= 1) {
16                d_sahEvents[i].sahValue = UNUSED;
17          }
18          else {
19
20                unsigned int initialEventPos = d_nodeInitialPos[nodeNumber];
21                unsigned int lastEventPos = initialEventPos + (nodeNumTriangles << 1) - 1;
22
23                float xBot = d_eventX[initialEventPos];
24                float xTop = d_eventX[lastEventPos];
25                float yBot = d_eventY[initialEventPos];
26                float yTop = d_eventY[lastEventPos];
27
28                float fatherSurfaceArea = calculateSurfaceArea(xTop, xBot, yTop, yBot);
29                float leftSurfaceArea, rightSurfaceArea;
30                unsigned int posStart, posEnd;
31
32                // SAH para eventos en X
33                posStart = d_posXStart[i];
34                float eventXStart = d_eventX[posStart];
35                leftSurfaceArea = calculateSurfaceArea(eventXStart, xBot, yTop, yBot);
36                rightSurfaceArea = calculateSurfaceArea(xTop, eventXStart, yTop, yBot);
37                float sahXStart = calculateSAH(...);
38                // repetición para el evento End
39                ...
40                // SAH para eventos en Y
41                posStart = d_posYStart[i];
42                float eventYStart = d_eventY[posStart];
43                leftSurfaceArea = calculateSurfaceArea(xTop, xBot, eventYStart, yBot);
44                rightSurfaceArea = calculateSurfaceArea(xTop, xBot, yTop, eventYStart);
45                float sahYStart = calculateSAH(...);
46                // repetición para el evento End
47                ...
48                storeMin(...); // Guarda el mejor de los 4 candidatos para esta primitiva
49          }
50    } }
```

**Algoritmo 4.5 Código para la function de coste de la SAH.**

```
00 __device__ float SAHEvent(
01    float*          d_event,            unsigned int*   d_scannedFlags,
02    unsigned int    offset,             float           eventT,
03    float           initialEvent,       float           lastEvent,
04    float           leftSurfaceArea,    float           rightSurfaceArea,
05    float           fatherSurfaceArea,  unsigned int    nodeNumPrimitives,
06    unsigned int    eventPos,           unsigned int    initialEventPos)
07 {
08    float sahValue;
09
10    if ( (initialEvent == eventT) || (lastEvent == eventT) )
11          sahValue = UNUSED;
12    else {
13          unsigned int scannedFlag = d_scannedFlags[eventPos + offset] -
14                d_scannedFlags[initialEventPos + offset];
15          float NL = (float) scannedFlag;
16          float NR = (float)(nodeNumPrimitives - (eventPos - initialEventPos -
17                scannedFlag));
18          // Sah(event) = 1 + ( (NL * AABBL + NR * AABBR) / AABBFather )
19          sahValue = 1.0f + ( ( (NL * leftSurfaceArea) + (NR * rightSurfaceArea) )
20                      / fatherSurfaceArea);
21          if (sahValue > nodeNumPrimitives)
22                sahValue = UNUSED;
23    }
24    return sahValue;
25 }
```

### *Obtención de las mínimas SAH*

El mejor candidato seleccionado por cada primitiva se guarda en un array (d_sahValues) como el de la Figura 4.7. Este array contiene en cada elemento, el valor SAH, el evento asociado a esa SAH, y su eje. Usando el array que contiene el nodo al que pertenece cada primitiva (d_nodeForPrimitive) se ejecuta la operación de reducción segmentada para hallar el mínimo de cada nodo. Con el elemento mínimo, se guarda el evento (en d_nodeSplittingRect) y el eje (en d_nodeAxis). En la Figura 4.7, suponiendo que para el nodo 1 el mínimo valor de SAH es de T2, se escribe en d_nodeSplittinRect y d_nodeAxis el evento y el eje de T2. Del mismo modo, suponiendo que el mínimo valor de SAH para el nodo 2 procede de T3, se escribe el evento y el eje del mejor valor de SAH obtenido para T3.

Como detalle de implementación, THRUST no es capaz de calcular mínimos para un tipo definido por el usuario (como el del array d_sahValues). Es necesario especificarle un puntero a una función que realice esa comprobación entre dos elementos como en el Algoritmo 4.6.

**Figura 4.7 Reducción segmentada para el cálculo de mínimos por nodo.**

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| d_nodeForPrimitive | 1 | 1 | 2 | 2 |
| d_sahValues.sah | sah1 | sah2 | sah3 | sah4 |
| d_sahValues.event | event1 | event2 | event3 | event4 |
| d_sahValues.axis | axis1 | axis2 | axis3 | axis4 |

⇩ Reducción segmentada para calcular mínimos

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeSplittingRect | | event2 | event3 | | | | |
| d_nodeAxis | | axis2 | axis3 | | | | |

**Algoritmo 4.6 Definición del tipo empleado para almacenar los valores SAH y la función que necesita THRUST para comparar dos elementos y obtener el mínimo.**

```
typedef struct key_value {
    float           sahValue;
    float           eventValue;
    unsigned int    axis;
};

typedef struct min_key_value : public
    thrust::binary_function<key_value&,key_value&,key_value> {
    __host__ __device__
        key_value operator() (key_value &lhs, key_value &rhs) const
    {
        return lhs.sahValue < rhs.sahValue ? lhs : rhs;
    }
};
```

## División de nodos

En la versión 2D, la división de nodos consta de las siguientes fases:

- Generación de flags. Hay que determinar a qué hijo debe ir cada primitiva con la recta de corte elegida para su nodo.
- Aplicación de *clipping*. Clipping es una técnica que recorta una figura. Para las primitivas que son cruzadas por la recta de corte, se emplea el clipping para determinar las nuevas cajas del hijo izquierdo y derecho.
- Primera fase de reordenamiento. El clipping produce desorden en las listas de eventos y se dan los primeros pasos para una ordenación eficiente.
- Preparación de las estructuras para la siguiente iteración. Del mismo modo que en la versión 1D, es necesario escribir los resultados de la iteración actual en otra estructura.
- Segunda fase de reordenamiento. Se termina la ordenación de las listas de eventos en paralelo en la estructura preparada para la siguiente iteración.
- Maximización del espacio vacío. Es una técnica que mejora la calidad del árbol generado, reduciendo el tamaño de las cajas para los nuevos nodos generados.

A continuación se explican cada una de estas fases en detalle.

### *Generación de flags*

En 2D, dada una primitiva y una recta se pueden producir los casos que muestra la Figura 4.8. Puede quedar completamente a la izquierda o a la derecha de la recta, o ser cruzada. Por un lado, la versión 1D lanza un hilo por evento, los eventos end acceden al evento start que les corresponde y comprueban en qué situación se encuentra el segmento respecto al punto de corte. El problema de ese enfoque es que los hilos de los eventos start no realizaban ninguna tarea, desaprovechando los recursos de la GPU. Sin embargo, en la versión 2D se lanza un hilo por triángulo. El kernel del Algoritmo 4.7, implementa esta fase. Empieza accediendo al nodo asociado a la primitiva para acceder a la recta de corte y su eje (líneas 12-14). Después guarda en variables locales las posiciones de sus cuatro eventos (líneas15-16). Del mismo modo que para 1D, se comienza suponiendo que la primitiva será duplicada (líneas 18-20), es decir cruzada por la recta de corte, y luego se comprueba si efectivamente es así.

**Algoritmo 4.7 Kernel para la generación de flags.**

```
00 __global__ void KernelGenFlags(
01    unsigned int*   d_count,              unsigned int*    d_childSide,
02    unsigned int*   d_primitiveNode,      unsigned int*    d_posXStart,
03    unsigned int*   d_posXEnd,            unsigned int*    d_posYStart,
04    unsigned int*   d_posYEnd,            float*           d_eventX,
05    float*          d_eventY,             float*           d_nodeSplittingRects,
06    unsigned int*   d_cuttingPlaneAxis, unsigned int       numPrimitives,
07    unsigned int    numEvents)
08 {
09    unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
10    // Para cada primitiva
11    if (i < numPrimitives)  {
12          unsigned int nodeNumber = d_primitiveNode[i];
13          float rect = d_nodeSplittingRects[nodeNumber];
14          unsigned int axis = d_nodeSplittingAxis[nodeNumber];
15          unsigned int xStartPos = d_posXStart[i], xEndPos = d_posXEnd[i];
16          unsigned int yStartPos = d_posYStart[i], yEndPos = d_posYEnd[i];
17          // Se comienza suponiendo que es duplicado
18          unsigned int childFlag = DUP_SIDE;
19          unsigned int leftPrimitiveFlag = 0, rightPrimitiveFlag = 0;
20          unsigned int leafPrimitiveFlag = 0;
21          // Si no engendra, se convierte en parte de una hoja
22          if (rect == UNUSED) {
23                // Los nodos hojas van la zona de nodos hoja
24                childFlag = LEAF_SIDE;
25                leafPrimitiveFlag = 1;
26          }
27          else {
28                if (axis == X_AXIS) { // Si el corte es en el eje X
29                      // Se cogen el start y el end de X
30                      float eventStart = d_eventX[xStartPos];
31                      float eventEnd = d_eventX[xEndPos];
32                      if (eventEnd <= rect) { // Va al lado izquierdo:  Start----End  |
33                            childFlag = LEFT_SIDE; // Se marca que está a la izquierda
34                            leftPrimitiveFlag = 1;
35                      }
36                      // Puede ser un duplicado, es necesario mirar  su start
37                      else if (rect  <= eventStart) {  // |   Start----End
38                            childFlag = RIGHT_SIDE; // Se marca que está a la derecha
39                            rightPrimitiveFlag = 1;
40                      }
41                            // si no, es duplicado: Start--|--End
42                }
43                else { // El eje de corte es el eje Y
44                      // Se hace lo mismo que se hace con el eje X
45                      // con los eventos del eje Y
46                }
47          }
48          d_childSide[i] = childFlag;
49          // Inicialización de flags para reordenamiento en d_count
50          // usando leftPrimitiveFlag y rightPrimitiveFlag
51          ...
52 } }
```

La diferencia respecto a la versión 1D es que hay que determinar tres datos o flags. El primero de ellos se escribe en d_childSide, (línea 18) y puede tener cuatro valores: lado izquierdo (LEFT_SIDE), lado derecho (RIGHT_SIDE), duplicado (DUP_SIDE) o va a formar

parte de un nuevo nodo hoja (LEAF_SIDE). La comprobación de a qué tipo pertenece la primitiva se realiza de forma similar que para 1D, solo que en esta ocasión hay que considerar el eje en el que se produce el corte (línea 28).

El siguiente tipo de flag se escribe en las primeras numPrimitives posiciones en d_count (líneas 49-50) (ver Figura 4.9). En él, usando el flag d_leafPrimitiveFlag (línea 20), cada primitiva escribe un 1 si se convierte en nodo hoja, y 0 en caso contrario. Tras hacer un scan exclusivo, se podrá establecer la posición en la que escribir las primitivas de los nodos hojas en la estructura que los almacenará para la solución final.

Y con el tercer tipo de flag, cada primitiva determina para cada uno de sus cuatro eventos (línea 19) si pertenecen al hijo izquierdo con d_leftPrimitiveFlag (escribiendo un 1 en Izq X e Izq Y de la Figura 4.9) y si pertenecen al hijo derecho con d_rightPrimitiveFlag (escribiendo un 1 en Der X y Der Y de la Figura 4.9). En la versión 1D, los eventos duplicados escribían un 1 tanto en el hijo izquierdo como en el derecho porque se copian a los dos hijos. En esta versión, los duplicados van a escribir un 0 inicialmente en las cuatro partes (Izq X, Izq Y, Der X y Der Y). Para entender por qué, es necesario comprender los efectos producidos por el clipping.

**Figura 4.8 Situaciones que pueden ocurrir entre una recta y una primitiva (a la izquierda, cruzada y a la derecha).**

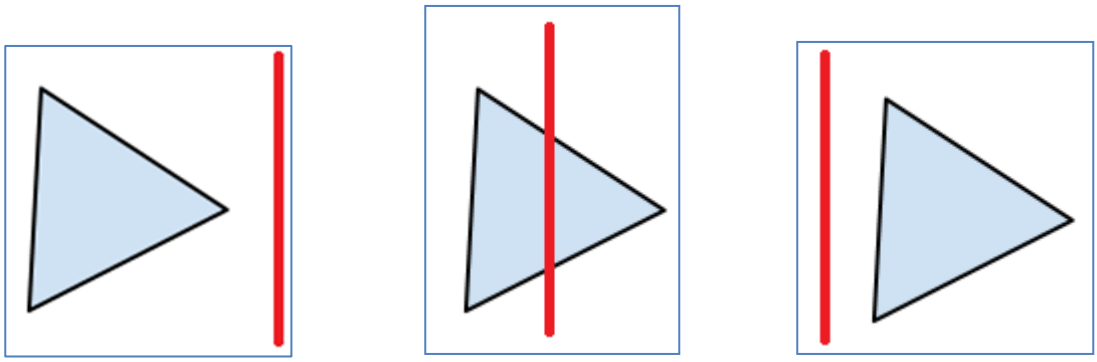
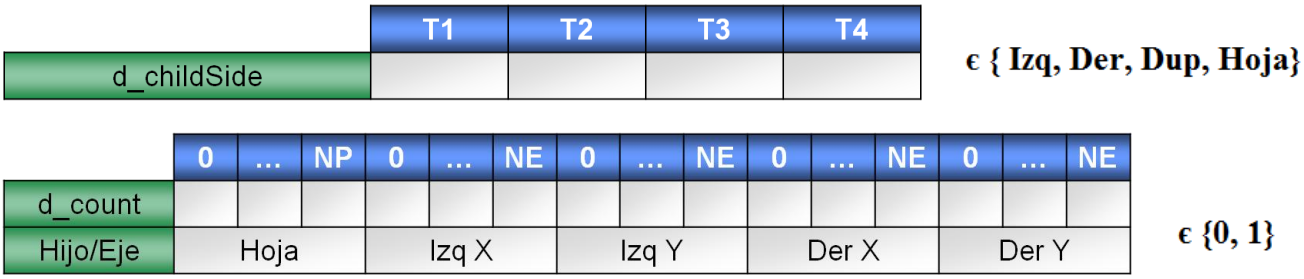


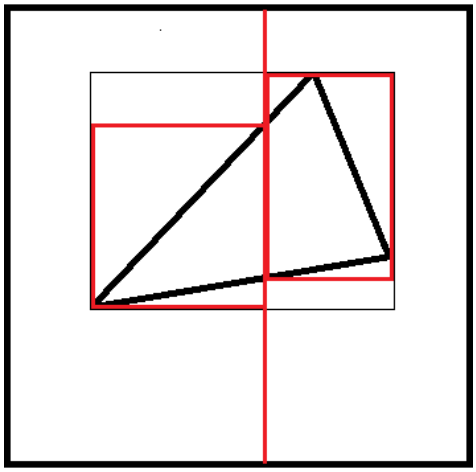
**Figura 4.9 Estructuras para almacenar flags. Para cada primitiva, d_childSide, un enumerado para su situación respecto a la recta de corte. En la primera parte de d_count, cada primitiva escribe un 1 si es hoja o 0 en caso contrario. En las demás partes (Izq X, Izq Y, Der X y Der Y), cada primitiva escribe para cada uno de sus eventos un 1 si la primitiva pertenece a ese hijo.**




### *Clipping*

La Figura 4.10 muestra un triángulo partido por una recta del tipo X = cte. Mediante clipping se obtienen las cajas de los hijos. Para el eje Y, una opción sería mantener los límites de la caja del padre, pero calculando las cajas mínimas que envuelven a los hijos mejora la calidad del árbol generado. Para ray tracing, la eliminación de ese espacio vacío mejora la eficiencia porque si un rayo está en el espacio vacío, no tiene que comprobar intersección con primitivas. Por lo tanto, mediante clipping se obtienen las nuevas cajas para los hijos.

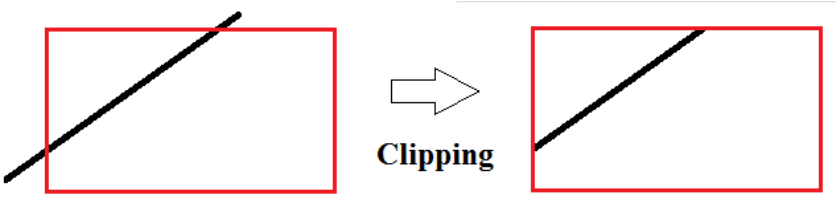
**Figura 4.10 Clipping para un triángulo partido por una recta de corte en el eje X y que genera dos cajas en las que varían los límites en el eje Y.**



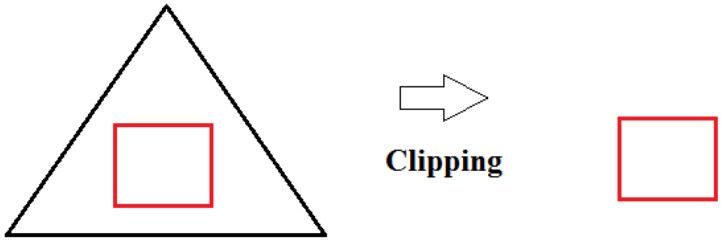
La obtención de los nuevos límites de las cajas de los hijos se realiza mediante un algoritmo de recorte de segmentos, conocido como “Cyrus-Beck”. En nuestro caso, el algoritmo tomará un rectángulo alineado con los ejes y un segmento, y lo recortará obteniendo la parte del segmento contenida en el rectángulo. Un ejemplo de este resultado es el de la Figura 4.11 (a).

En la Figura 4.12 se muestra un ejemplo para el recorte de un segmento con extremos (X1, Y1) y (X2, Y2) frente a la frontera izquierda (la recta originada por el lado izquierdo del rectángulo). En la figura, se calcula la coordenada Y para el punto de corte que sucede en la frontera X = eventXStart. Una vez conocido el nuevo valor de Y, se actualizan los valores de X1 e Y1 y continúa el recorte con las otras 3 fronteras. Es importante distinguir el caso en el que el segmento es paralelo a algún lado de la caja, o que el segmento esté totalmente fuera de la caja, en cuyo caso se descarta el segmento al completo.

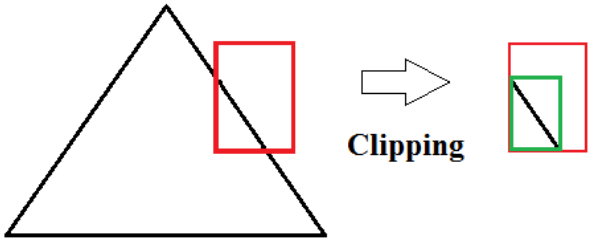
**Figura 4.11 (a) Clipping para un segmento con un rectángulo. (b) Caso degenerado de clipping en el que no queda ningún segmento tras el recorte. (c) Clipping en el que solo queda tras el recorte un lado del triángulo.**




(a)




(b)



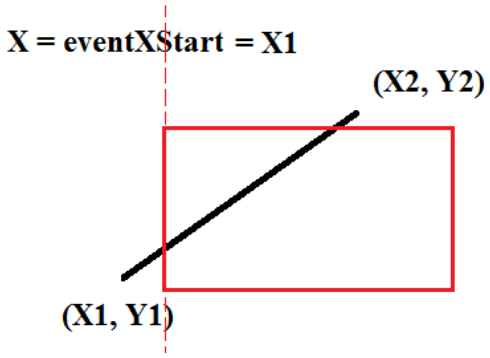

(c)

El proceso de recorte se realiza para cada uno de los segmentos que forman el triángulo, tanto para el hijo izquierdo como para el hijo derecho que resultarían de la recta de corte candidata. Una vez se tienen los segmentos contenidos dentro de la caja de cada hijo, se pueden calcular los nuevos límites, tratando de manera especial casos degenerados. Pueden suceder tres casos:

- Que no quede ningún segmento dentro de la caja (Figura 4.11 b). Se mantienen los mismos límites que para el padre teniendo en cuenta la recta de corte.
- Que quede un segmento dentro de la caja (Figura 4.11 c). Hay que determinar qué región es la que interseca con el triángulo y cuál no. Se comprueba mirando dónde hay más puntos originales del triángulo. Por ejemplo, en la Figura 4.11 c, respecto al eje Y, hay 2 puntos en la parte inferior y uno en la superior. Respecto al eje X, hay 2 puntos a la izquierda y uno a la derecha. Como conclusión, la parte que pertenece al triángulo es la región inferior-izquierda, y los límites son los marcados en verde.
- Que queden dos o tres segmentos (Figura 4.10). Es suficiente con calcular máximos y mínimos.

Como muestra la Figura 4.10, el clipping genera nuevos valores para los eventos en el eje Y, y esto puede originar desorden en los eventos del eje Y. En el eje X se sigue respetando el orden entre eventos. En el eje en el que se produce el corte, permanecen inalterados, manteniéndose el orden que había antes. Por el contrario, en los otros ejes es necesario reordenar. Para evitar una ordenación completa y desde cero que haría inviable el algoritmo, se utiliza localidad espacial: los eventos que cambian de valor lo hacen dentro de los límites de la caja del padre. Por lo tanto, los límites de la caja del padre son los límites para la búsqueda de la nueva posición de estos eventos modificados.

**Figura 4.12 Obtención de puntos de corte para recortar segmentos con Cyrus-Beck.**




Ecuaciones de la recta negra:

$$I: x = (x_2 - x_1)\, t + \, x_1$$

$$II: y = (y_2 - y_1)\, t + \, y_1$$

Despejando t en I:

$$t = \frac{x - \, x_1}{x_2 - x_1}$$

El punto de corte para la frontera izquierda está en X = eventXStart, por lo que queda:

$$t = \frac{eventXStart - \, x_1}{x_2 - x_1}$$

Después se calcula el valor de y en II:

$$y = (y_2 - y_1)\, t + \, y_1$$

### *Primera fase de reordenamiento*

El método de ordenación de los arrays de eventos se basa en el algoritmo "*bucket-sort"*. Este algoritmo construye intervalos en función de los eventos. La Figura 4.13 muestra un ejemplo de eventos y sus intervalos asociados. El array clipped contiene los resultados de los eventos modificados por el clipping.

El primer paso para la ordenación consiste en clasificar los eventos en los intervalos construidos. Para los eventos que no se han modificado, es decir, aquellos que no son resultado del clipping, se les asigna directamente el intervalo al que hacen referencia. Por ejemplo, en la Figura 4.13, todos los eventos que no se modifican tienen asignado en d_bucket la posición del intervalo que le pertenece, que coincide con su posición en d_eventX. Además, durante la fase de generación de flags se escribió en el array d_count un 1 para los eventos no duplicados en la

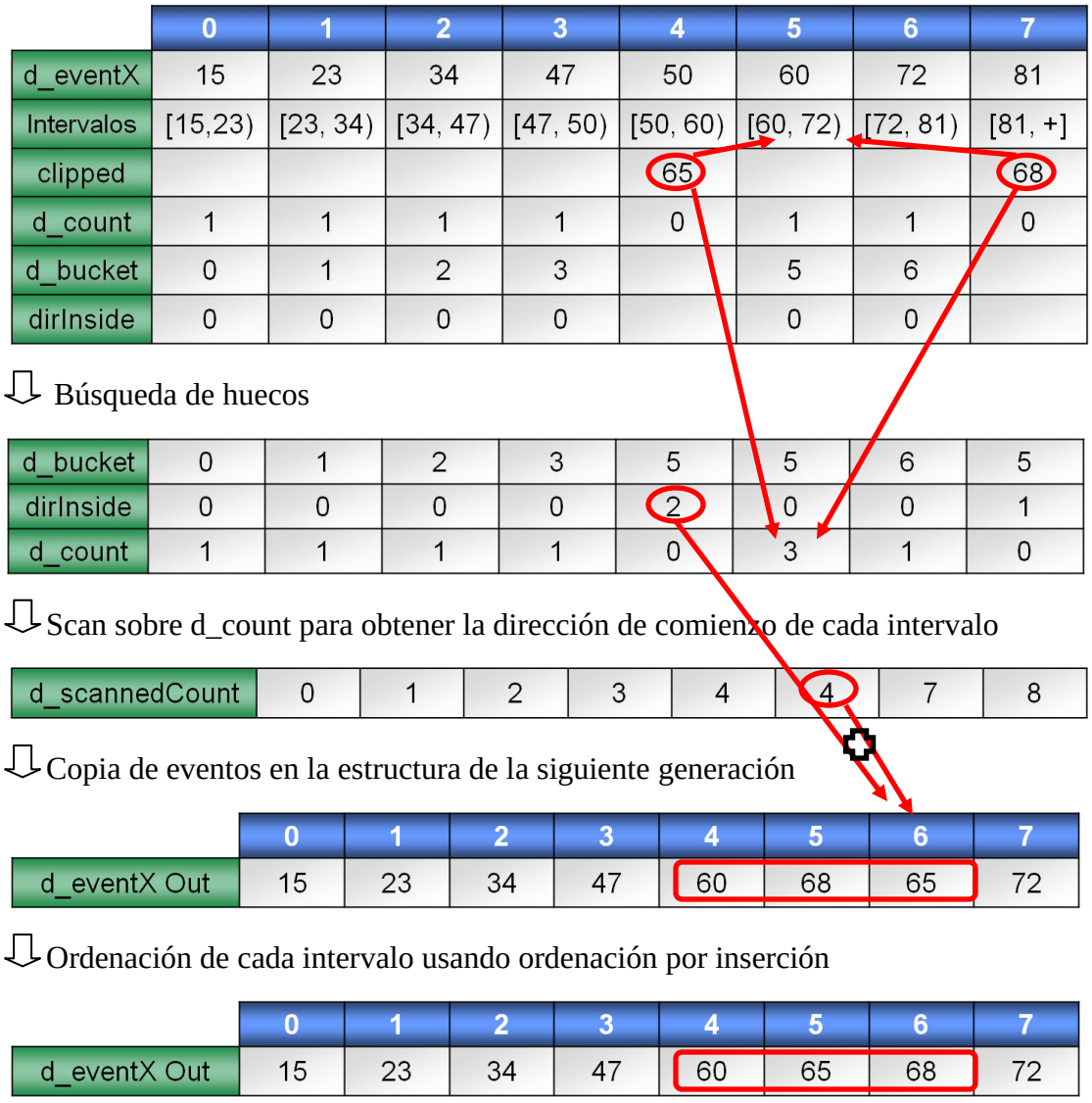
parte que les corresponde del array d_count según lo explicado en esa sección (Izq X, Izq Y, DerX y Der Y), y toman como dirección dentro del intervalo (dirInside), la posición 0.

Para los eventos de primitivas duplicadas, como los de las posiciones 4 y 7, se escribió un 0 en d_count. Ello se debe a que el clipping modificaría esos eventos, haciendo necesario buscar los intervalos que les corresponde a los eventos modificados y desechar los antiguos valores. En esta fase se emplea búsqueda binaria para encontrar esos intervalos, y está limitada en el rango de los valores de los eventos de partida. Por ejemplo, en la Figura 4.13, los eventos de las posiciones 4 y 7 son duplicados. Después de la generación de flags y el clipping, se busca el intervalo que les corresponde a los eventos modificados 65 y 68, que es el del intervalo [60-72) para ambos (los límites de la búsqueda binaria son los antiguos valores en d_eventX, 50 y 81). Para estos dos eventos, se determina que su intervalo es el 5 (en d_bucket). En cuanto al array dirInside, sirve para almacenar la posición que tiene cada evento en su intervalo. Cuando un evento encuentra su intervalo, lee el contador que hay en él (en d_count). Supongamos que el primer evento que lee en d_count es el 68, y toma el 1 escrito en d_count para la posición 5. Ese 1 es la dirección que toma dentro del intervalo, y después incrementa el contador (subiéndolo a 2). Para realizar la lectura e incrementar el contador sin que otro evento interfiera, es necesario emplear una operación atómica. A continuación, llega el evento 65, lee un 2 en d_count y aumenta el contador a 3. Ese 2 es la posición dentro del intervalo.

Finalmente, todos los eventos quedan clasificados en los intervalos. En d_count se obtiene el número de eventos que contiene cada intervalo, y para cada evento, en d_bucket el índice del intervalo y en dirInside la posición dentro de él.

A continuación, se realiza la operación scan exclusivo de THRUST sobre el array d_count para obtener la dirección de comienzo de cada intervalo en la estructura de entrada de la siguiente generación.

**Figura 4.13 Búsqueda de intervalos para eventos, copia en la estructura de la siguiente generación y última fase de reordenación.**




### *Preparación de las estructuras para la siguiente iteración*

La siguiente fase es la copia de eventos en la estructura de swap. A cada evento, i, le corresponde d_scannedCount[ d_bucket[ i ] ] + dirInside[ i ], es decir, la dirección de comienzo de su intervalo, más la dirección dentro de él. Por lo tanto, la copia es inmediata. Además, esa posición se emplea para determinar los valores de los arrays d_posXStart, d_posXEnd, d_posYStart y d_posYEnd, según el tipo de evento que se esté copiando.

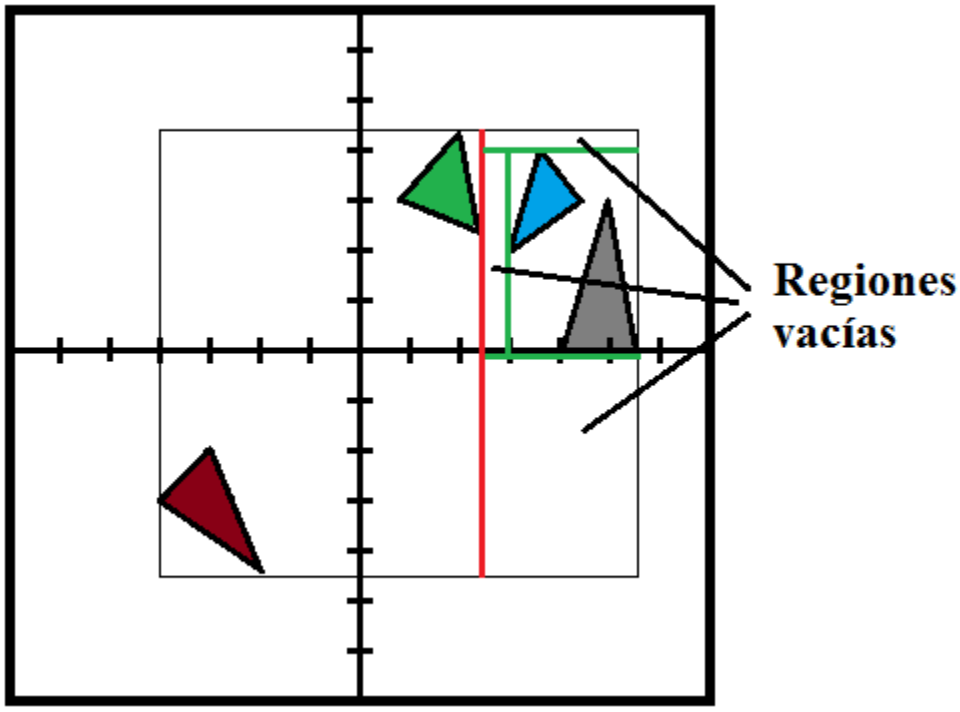
### *Segunda fase de reordenamiento*

Tras la copia, no necesariamente se mantienen ordenados los arrays de eventos, como sucede en el ejemplo de la Figura 4.13. En la penúltima tabla están marcados en rojo los eventos que proceden del intervalo 5. Es necesario un paso más para terminar. Lanzando un hilo por cada intervalo, cada intervalo se ordena por inserción. Suelen ser intervalos con pocos elementos, haciendo viable esta elección.

### *Maximización del espacio vacío*

La última técnica de la división de nodos se llama "Maximización del espacio vacío". Al hacer cortes quedan regiones de espacio vacío en las cajas generadas para cada hijo, como en el ejemplo de la Figura 4.14. Usando el menor y el mayor evento de las listas de eventos ordenados, se obtienen cajas que se ajustan mejor a los nodos de la nueva generación. Este enfoque mejora la eficiencia de ray tracing por la misma razón argumentada para el clipping, los rayos que caen en esos espacios vacíos no comprueban intersección con primitivas y fue propuesta por Havran [Hav00].

**Figura 4.14 Ejemplo de creación de listas de regiones vacías para maximización del espacio vacío tras hacer un corte (en rojo). Las líneas en verde indican los cortes generados para las regiones vacías.**



## Finalización

Cuando finaliza el bucle del algoritmo principal, se copia a la memoria del host las estructuras necesarias para interpretar el kd-tree:

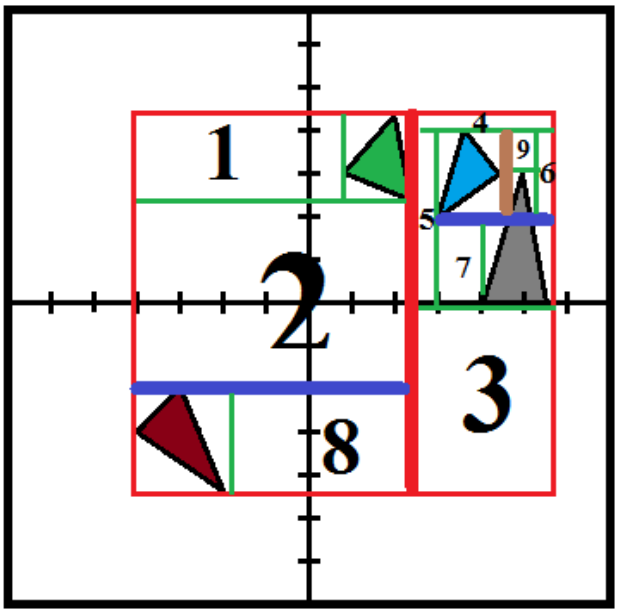
- El array con las listas de los índices de las primitivas que pertenecen a cada hoja (d_leafPrimitives).
- El array con la posición de comienzo de cada nodo hoja en el array de índices de primitivas (d_nodeInitialPos).
- Los arrays con la recta de corte (d_nodeSplittingRects) y el eje de cada nodo interno (d_nodeSplittingAxis).
- Número de primitivas que contiene cada nodo (d_nodeNumPrimitives) y las cajas que los envuelven (d_nodeAABBXMin/XMax/YMin/YMax).

Un ejemplo de estas estructuras y el árbol correspondiente son los de la Figura 4.15. En la imagen (a), los triángulos están rodeados por un rectángulo en rojo (AABB del nodo raíz). La recta de corte, en rojo grueso, deja los triángulos rojo y verde al lado izquierdo, y el azul y gris al lado derecho. Para el hijo derecho se crean las regiones vacías 3, 4 y 5. En la siguiente generación, la parte del hijo izquierdo del nodo raíz tiene como recta de corte la violeta en grueso y se generan las regiones vacías 1, 2 y 8. Para el hijo derecho del nodo raíz, con la recta en violeta grueso se crea un hijo en la parte inferior con la primitiva gris, y otro en la parte superior con la azul y una parte de la gris. Para ellos se crean las regiones vacías 6 y 7 respectivamente. Por último hay una nueva recta de corte en marrón grueso, que divide la región de la parte superior, dejando a un lado la primitiva azul y al otro la parte que quedaba de la gris, dando lugar a la región vacía 9.

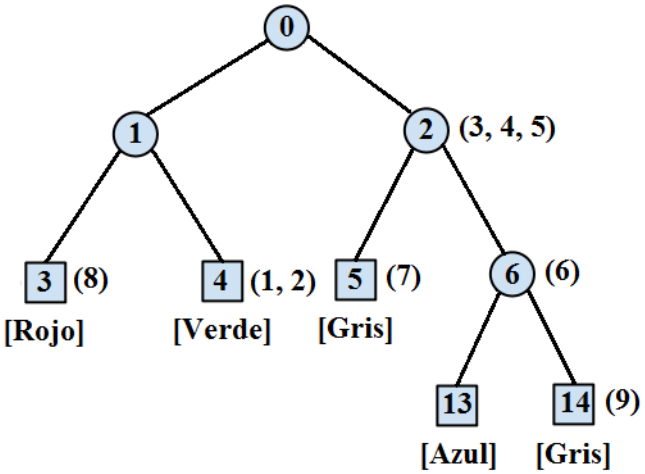
En la imagen b, se muestra el árbol resultante, cuyos nodos contienen el número de nodo, que coincide con la posición en los arrays de la tabla (c). Entre paréntesis se indican las regiones vacías asociadas a cada nodo según la numeración de la imagen a. En los nodos hoja (representados con un cuadrado), figura entre corchetes la primitiva que contienen.

En la tabla (c), el array d_splittingRects indica la recta de corte elegida para cada nodo, con el eje empleado en d_splittingAxis. El array d_leafPrimitives contiene los índices de los triángulos de todos los nodos hoja, y en d_nodeInitialPos, la posición de comienzo de cada nodo hoja en ese array.

**Figura 4.15 Ejemplo para mostrar el resultado final. En rojo, la caja del nodo raíz y su recta de corte (en grueso). En violeta las rectas de corte de los hijos del nodo raíz y en marrón la de uno de los descendientes. En verde las rectas que denotan regiones vacías, que están numeradas en cada nodo entre paréntesis.**




(a)




(b)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| d_splittingRects | 2.2 | -3 | 2 | | | | 4.3 | | | | | | | | |
| d_splittingAxis | X | Y | Y | | | | X | | | | | | | | |
| d_leafPrimitives | Rojo | Verde | Gris | Azul | Gris | | | | | | | | | | |
| d_nodeInitialPos | | | | 0 | 1 | 2 | | | | | | | | 3 | 4 |
| d_nodeNumPrimitives | 4 | 2 | 3 | 1 | 1 | 1 | | | | | | | | 1 | 1 |

(c)

# Capítulo 5 - Resultados experimentales

Para evaluar la calidad de los kd-trees producidos se han realizado tres tipos de experimentos: el primero mide características de los árboles resultantes, el segundo mide el tiempo necesario para la construcción de kd-trees en GPU de escenas con diferente número de triángulos en comparación con la construcción secuencial en CPU (tanto para la versión 1D como la versión 2D/3D), y con el tercero se ha obtenido el porcentaje de tiempo de cada etapa del algoritmo 2D/3D con objeto de analizar cuáles tienen mayor impacto y planear su mejora en el futuro. La máquina empleada para las pruebas es un Intel Xeon E5645 (dos núcleos) con 24 GB de memoria RAM y tarjeta NVidia Tesla C2075 con 6 GB de memoria GDDR5.

## Características de los kd-trees

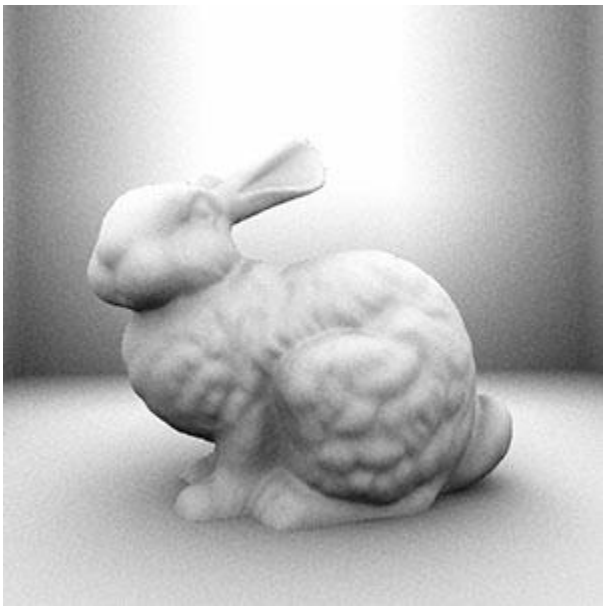
En la primera prueba se han empleado tres escenas en 3D comúnmente usadas por otros trabajos de construcción de kd-trees (mostradas en la Figura 5.1), dos incluidas en el repositorio de la Universidad de Stanford (Conejo, Armadillo) y otra descargada de un portal web (Sala de conferencias). También he desarrollado un generador de segmentos y triángulos pseudo-aleatorio, empleando la función "random" de la librería "time" C, que toma como uno de los parámetros de entrada el número de triángulos que tiene que generar. Los otros parámetros determinan los límites de la escena, los cuales han sido establecidos en -5.000 y 5.000 para cada dimensión.

La Tabla 5.1 muestra los resultados que se han obtenido para cada una de las escenas, así como las características de una escena con 100.000 triángulos en 3D construida de forma aleatoria.

Como característica de los árboles binarios, los nodos internos generados son uno menos que los nodos hoja. Las regiones generadas con espacio vacío son referenciadas desde los nodos del árbol, y determinan el espacio vacío generado por la técnica de "Maximización del espacio vacío" explicada al final del capítulo 4, en la construcción de kd-trees para la versión 2D. Es un número de regiones muy elevado en comparación con los nodos internos, ya que la generación de estas regiones se produce por el espacio vacío generado en los ejes X, Y o Z del hijo izquierdo y derecho al realizar un corte para un nodo interno, produciendo de 1 a 6 regiones de espacio vacío en 3D. En cuanto a las referencias, cada nodo hoja tiene el listado de los triángulos que

contiene su región. El número de referencias indicado en la tabla es la suma del tamaño de la lista de cada nodo hoja. Este número aumenta según se producen más triángulos duplicados por los planos de corte. Por ejemplo, para la escena "Sala de conferencias", se producen muchos duplicados y el número de referencias es superior que la escena del "Armadillo", aunque esta última tenga más triángulos. La media de referencias por hoja es un dato importante, ya que determina con cuántas primitivas hay que comprobar intersección en media para un rayo en "ray tracing" que llega a una hoja. Estos valores están influenciados por la SAH, que determina cuándo no hay ganancia con la división de nodos porque sus primitivas estén muy concentradas y al dividir se duplican la mayoría de ellas. Por último, la memoria requerida para contener los árboles generados, así como las estructuras para la construcción de los mismos, está influida por las predicciones realizadas antes de ejecutar el algoritmo. Éstas emplean el número de triángulos de la escena, con el que se hace la predicción del número de niveles del árbol y los tamaños para estructuras de nodos y eventos (reservando suficiente espacio para contener los eventos duplicados) como he explicado en la sección "Pasos previos" del capítulo 3. Hay que destacar que las estructuras para almacenar el árbol tienen capacidad para el árbol completo de esa profundidad, aunque el árbol generado por el constructor requiera menos espacio por no ser completo.

**Figura 5.1 Escenas empleadas para las pruebas: a) Conejo b) Armadillo y c) Sala de conferencias.**



(a)

(b)

(c)

A diferencia de las otras tres escenas, la escena realizada mediante generación automática distribuye los triángulos de forma aleatoria. En las otras tres escenas, al tratarse de objetos 3D, los triángulos están concentrados en ciertas regiones para delimitar los objetos. Esa concentración favorece la generación de triángulos duplicados, como resultado de los planos de corte. Por ello, la escena "Random" contiene prácticamente las mismas referencias que

triángulos de partida y ocupa mucho menos espacio que la escena del “Conejo” pese a tener más triángulos.

**Tabla 5.1 Tabla con las características de los kd-trees construidos en GPU (3D).**

| | Conejo | Conferencia | Armadillo | Random |
|---|---|---|---|---|
| Triángulos | 69 451 | 282 759 | 345 944 | 100 000 |
| Nodos | 330 471 | 289 867 | 561 697 | 53 791 |
| Nodos internos | 165 235 | 144933 | 314 848 | 26 895 |
| Nodos hoja | 165 236 | 144 934 | 314 849 | 26 896 |
| Regiones con espacio vacío | 381 890 | 252 163 | 667 461 | 143 178 |
| Profundidad | 18 | 20 | 20 | 18 |
| Referencias | 417 970 | 905 868 | 689 023 | 100 038 |
| Media Ref / hoja | 4.1 | 8.6 | 4.2 | 1.6 |
| Tamaño en memoria del kd-tree | 32.18 MB | 92.94 MB | 95.83 MB | 21.52 MB |
| Uso de memoria en GPU | 145.76 MB | 590 MB | 678.95 MB | 188.76 MB |

Además, he realizado una prueba de la corrección de los árboles construidos. Esta prueba consiste en tres etapas:

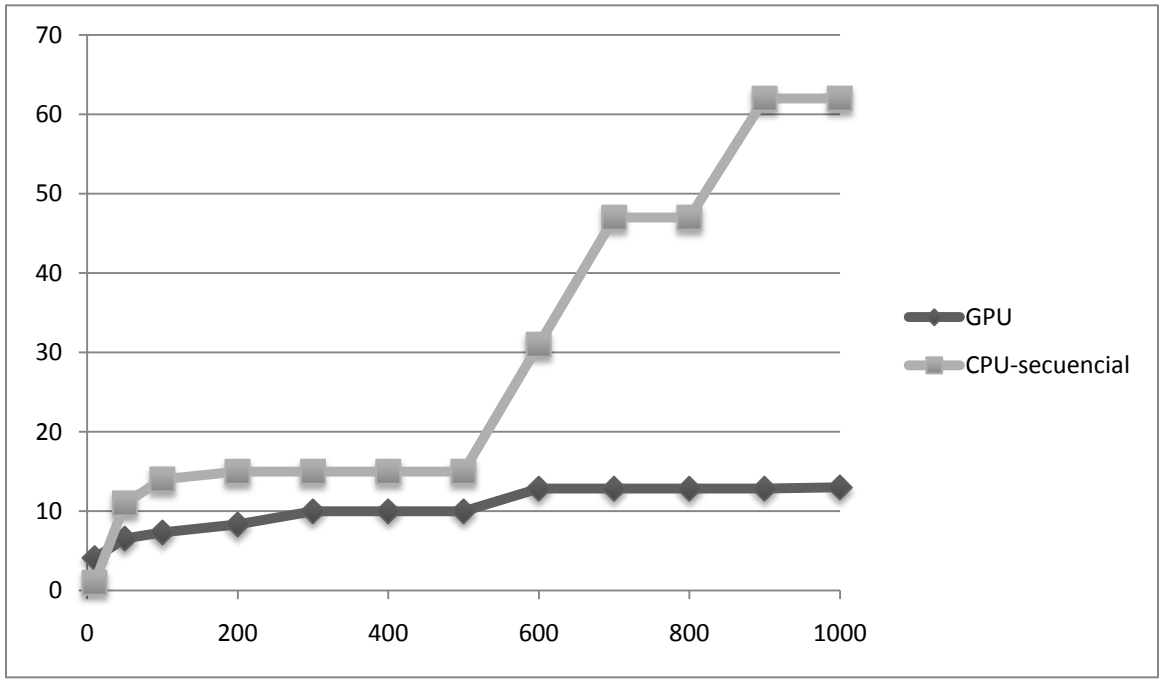
- Comprobar que se trata de un árbol binario, es decir, que cada nodo interno tiene dos hijos y los nodos hoja ninguno.
- Comprobar que cada triángulo de la escena es referenciado al menos una vez por las hojas del árbol.
- Para cada hoja, comprobar que sus triángulos están contenidos en las AABBs del nodo.

## Tiempos de construcción

### *Experimento (versión 3D)*

La segunda prueba ha consistido en medir el tiempo medio de ejecución para escenas generadas automáticamente con el generador pseudo-aleatorio, en función del número de triángulos que deben construirse. En la Figura 5.2 se comparan los tiempos (en ms) de ejecución que precisa la construcción en GPU con los que necesita la construcción en CPU para escenas de 10 a 1.000 triángulos. A partir de 500 triángulos, el tiempo de ejecución en CPU crece considerablemente en comparación con el de GPU. La versión en GPU se favorece de tener más datos con los que trabajar en paralelo, mientras que la versión en CPU tiene más carga secuencial. Para menos de 500 triángulos, la versión en CPU mantiene tiempos semejantes a la GPU, sólo mejorándola cuando hay menos de 50 triángulos.

**Figura 5.2 Gráfica comparativa de tiempos de ejecución (eje Y) en ms, frente a escenas con diferente número de triángulos (eje X) para construcciones en GPU y CPU-secuencial.**




En las Figuras 5.3 y 5.4 se muestran los resultados para escenas con 1.000 a 100.000 triángulos para la versión en CPU-secuencial y para la versión GPU respectivamente. La gráfica de la Figura 5.3 tiene el rango de valores del eje Y entre 0 y 300.000 ms. Para escenas con 100.000 triángulos resulta inviable construir kd-trees en CPU para su renderizado en tiempo real. Por el contrario, la gráfica de la Figura 5.4 tiene el rango de valores del eje Y entre 0 y 120 ms,

con 110 ms para escenas con 100.000 triángulos, permitiendo construir 4 kd-trees por segundo lo que se aproxima al renderizado en tiempo real.

**Figura 5.3 Gráfica comparativa de tiempos de ejecución (eje Y) frente a escenas con diferente número de triángulos (eje X) para construcciones en CPU-secuencial (versión 3D).**

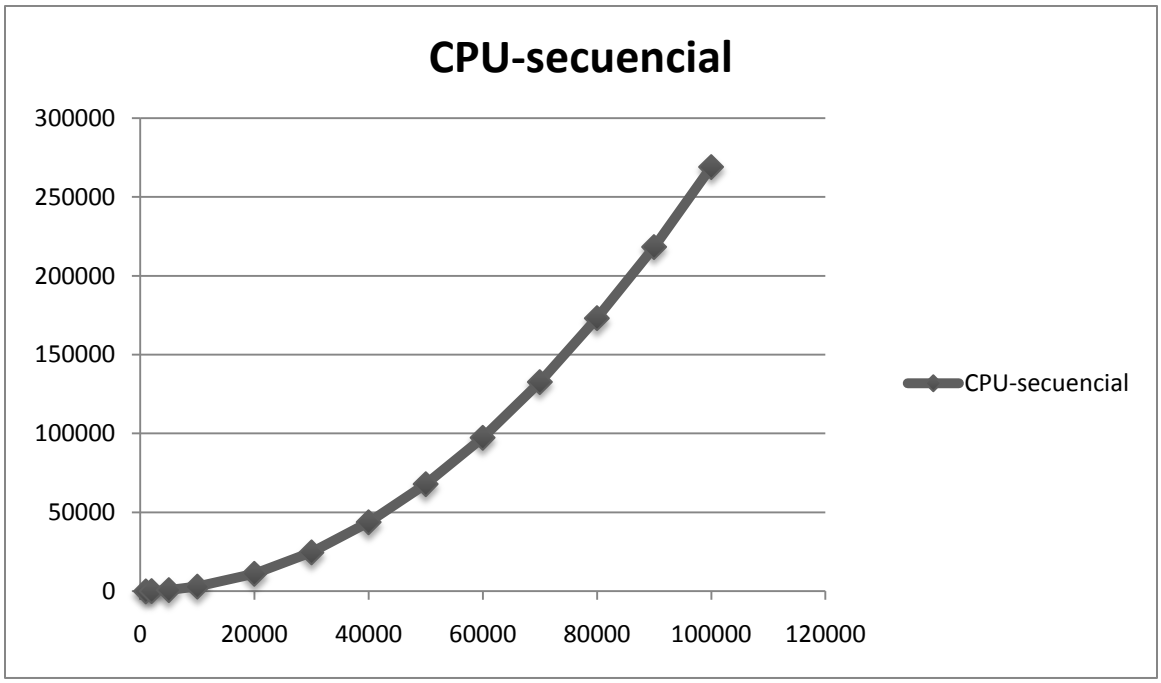
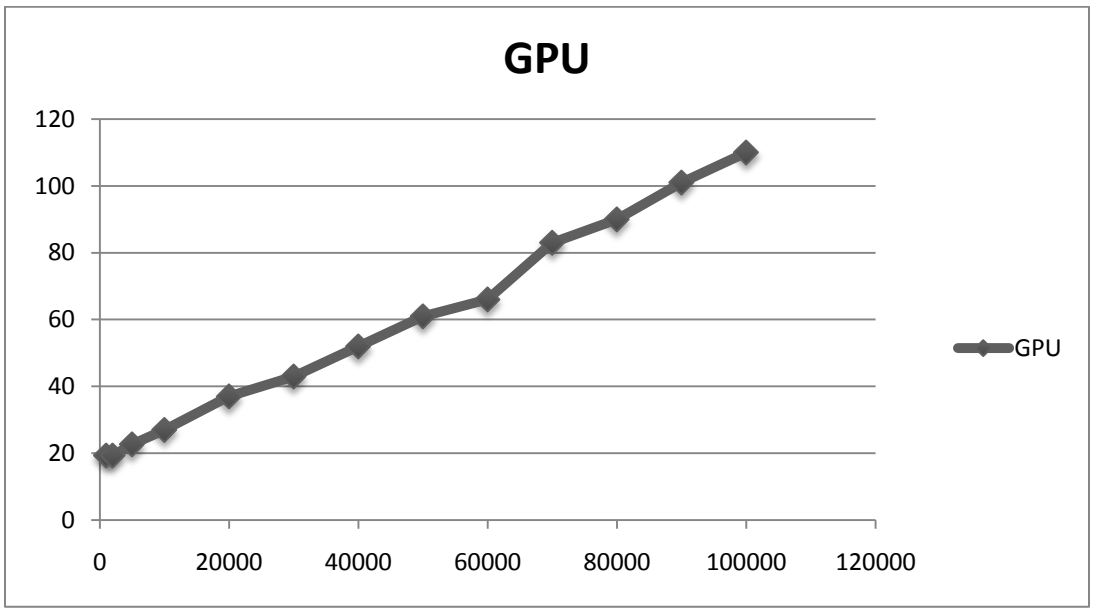



**Figura 5.4 Gráfica comparativa de tiempos de ejecución (eje Y) en ms, frente a escenas con diferente número de triángulos (eje X) para construcciones en GPU (versión 3D).**



Respecto a las tres escenas (“Conejo”, “Sala de conferencia” y “Armadillo”), la Tabla 5.2 muestra los tiempos de ejecución empleando la versión en GPU. Los tiempos para la versión en CPU-secuencial son muy elevados, 142 segundos para la escena del “Conejo” y más de una hora para la “Sala de conferencia” y el “Armadillo”. El tiempo de ejecución de la escena del “Conejo” es mayor que la escena creada automáticamente, por el motivo comentado anteriormente. La escena del “Conejo” tiene los triángulos que delimitan su superficie concentrados y, por el contrario, la escena “Random” tiene los triángulos dispersos, generando menos triángulos partidos por los planos de corte, haciendo por ello menos operaciones de clipping y ordenamiento.

**Tabla 5.2 Tabla con los tiempo de construcción en GPU de los kd-trees para las escenas de prueba.**

| | Conejo | Conferencia | Armadillo | Random |
|---|---|---|---|---|
| Triángulos | 69.451 | 282.759 | 345.944 | 100.000 |
| Tiempo | 112 ms | 309 ms | 420 ms | 105 ms |

***Experimento (versión 1D)***

Este experimento emplea una versión del generador para generar segmentos pseudo-aleatoriamente. La Figura 5.5 muestra los resultados obtenidos (en ms) para escenas con un número de segmentos comprendido entre 10 y 2000. En este caso, tanto la versión GPU como CPU obtienen tiempos de construcción menores que para el caso 3D, ya que solo es necesario ocuparse de una dimensión. Además, en la versión 1D es a partir de los 1000 segmentos en los que la GPU obtiene mejores resultados que la CPU por la secuencialización de la construcción. Sin embargo en 3D, la versión GPU es mejor incluso para escenas con menos de 500 triángulos que la versión CPU-secuencial. Esto se debe a que la versión GPU en 3D cuenta con una ventaja frente a la CPU que la versión 1D no tiene, y es que paraleliza la ejecución de las 3 dimensiones.

Las Figuras 5.6 y 5.7 muestran los comportamientos de las versiones en CPU-secuencial y GPU para un número de segmentos mayor. De nuevo, ambos resultados son tiempos menores que la versión 3D, ya que tratan solo una dimensión. La versión CPU-secuencial crece hasta tardar casi 25 segundos para 50.000 segmentos, mientras que la versión GPU tarda solo 37 ms.

**Figura 5.5 Gráfica comparativa de tiempos de ejecución (eje Y) frente a escenas con diferente número de triángulos (eje X) para construcciones en GPU y CPU-secuencial.**

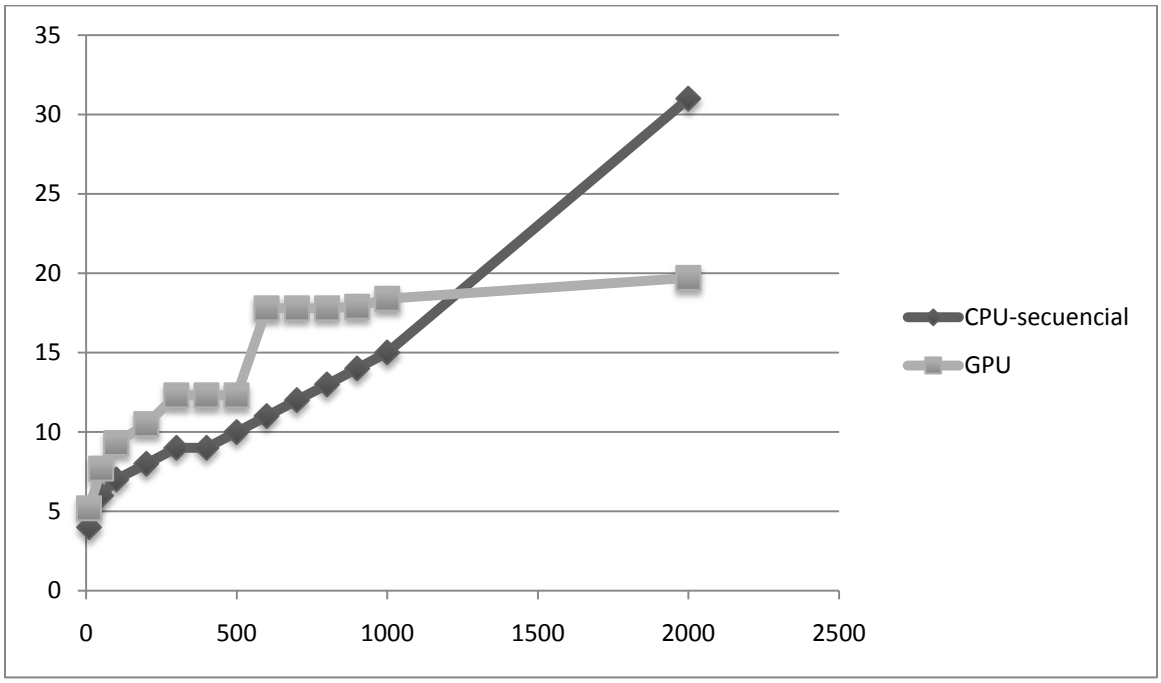



**Figura 5.6 Gráfica comparativa de tiempos de ejecución (eje Y) frente a escenas con diferente número de triángulos (eje X) para construcciones en CPU-secuencial (versión 1D).**

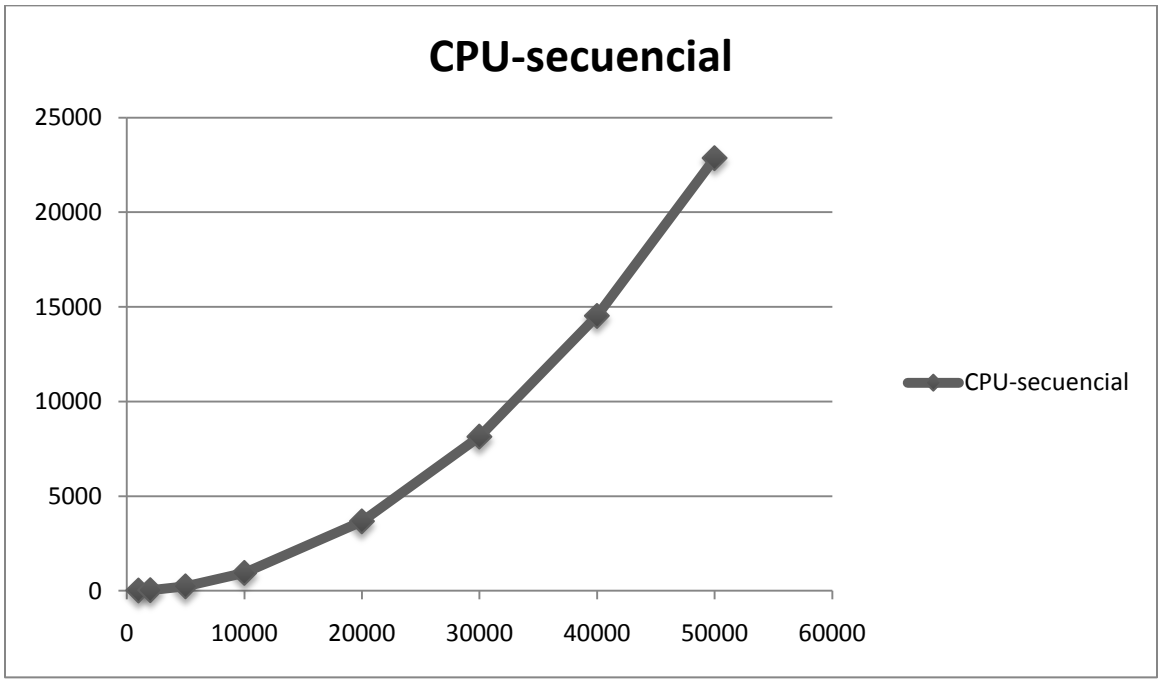
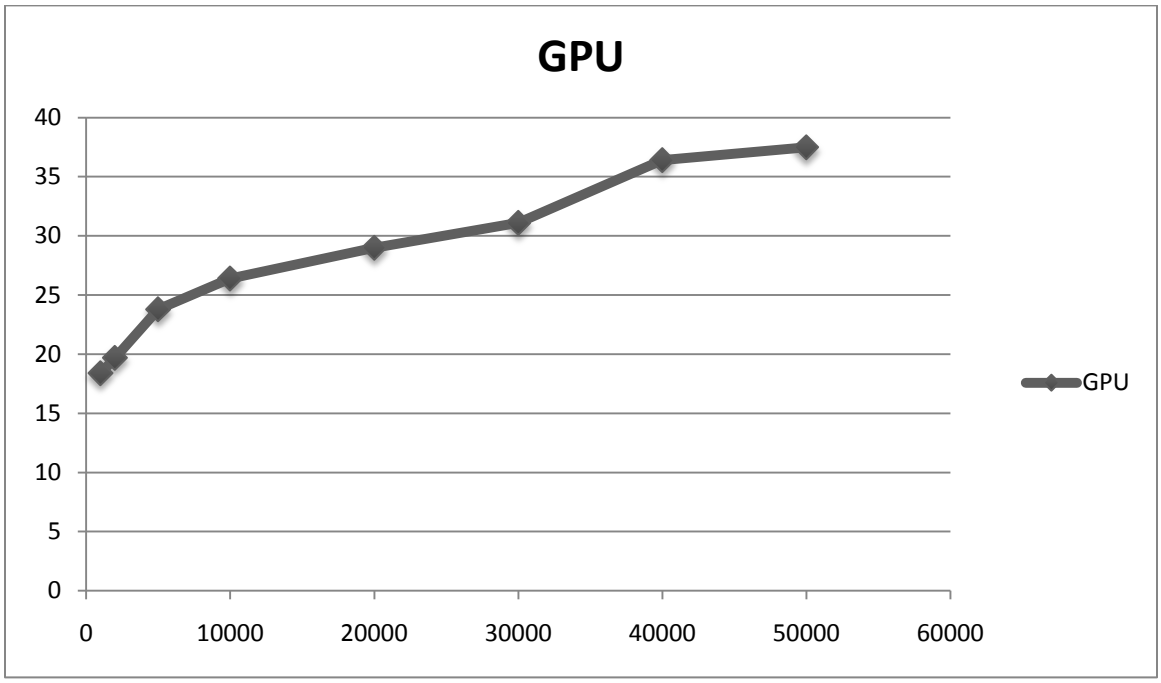


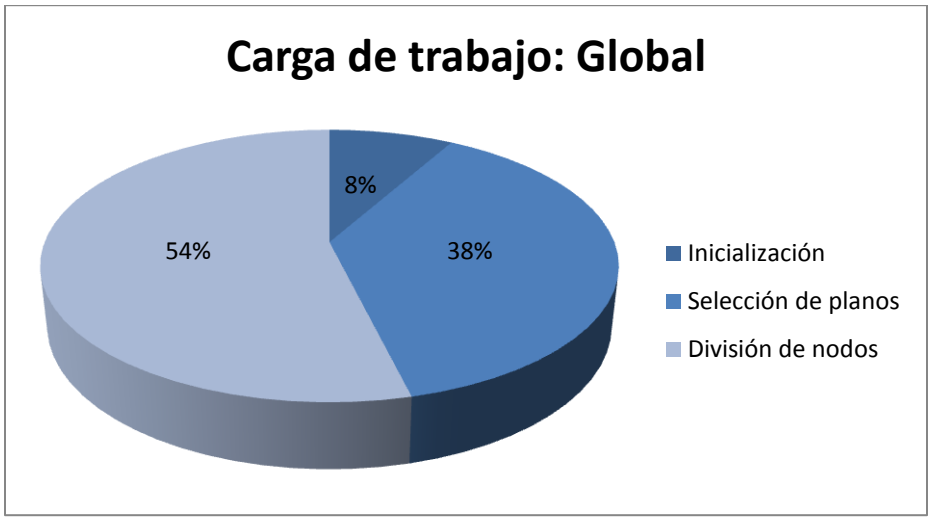
**Figura 5.7 Gráfica comparativa de tiempos de ejecución (eje Y) frente a escenas con diferente número de triángulos (eje X) para construcciones en GPU (versión 1D).**




## Reparto de trabajo

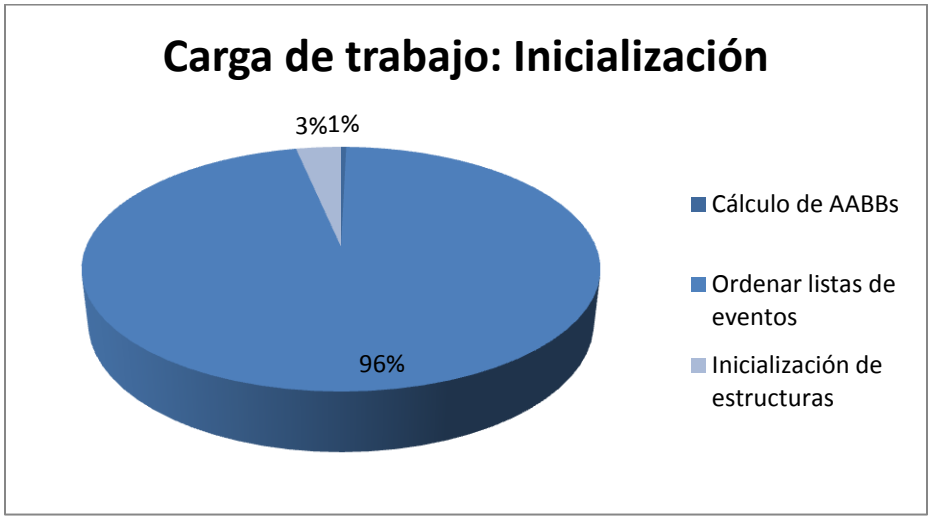
En la tercera prueba se ha medido el reparto del tiempo de ejecución de las tres fases del algoritmo de la versión 3D (inicialización, selección de planos de corte y división de nodos) para la escena del "Conejo". Esta prueba tiene como objetivos comprobar qué etapas del algoritmo son las que requieren mayor tiempo de ejecución y conocer en cuáles sería conveniente centrar esfuerzos en futuras mejoras para reducir impacto en el rendimiento. La Figura 5.8 muestra este reparto, en el que el mayor peso recae en la división de nodos. La inicialización, ya que se ejecuta una sola vez al comienzo del algoritmo, tiene sólo un 8%.

**Figura 5.8 Carga de trabajo para las etapas principales del algoritmo (versión 2D/3D).**



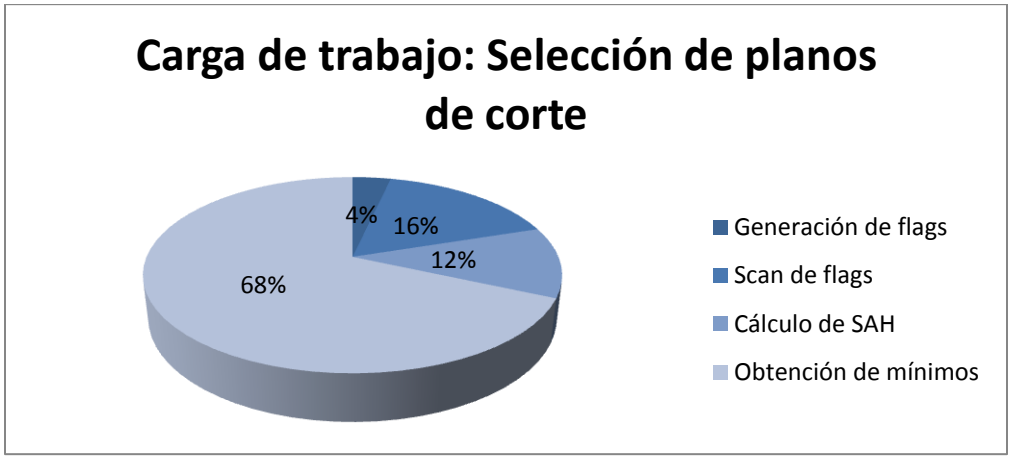

Para cada una de las fases que componen cada una de esas tres etapas también se ha medido el porcentaje de tiempo que consumen. Para la etapa de inicialización, la Figura 5.9 muestra los porcentajes del tiempo que se dedica al cálculo de las AABBs, la ordenación de listas de eventos y la inicialización de estructuras para nodos y eventos. En este caso, el peso recae en la ordenación de las listas de eventos, que se realiza mediante la operación "sort_by_key" de THRUST (una operación costosa). Las otras dos etapas, cálculo de AABBs e inicialización de estructuras, tardan tan poco en ejecutarse que para conseguir mejores tiempos en la etapa de inicialización habría que centrarse exclusivamente en la ordenación de las listas de eventos.

**Figura 5.9 Carga de trabajo para las etapas de la fase de inicialización.**



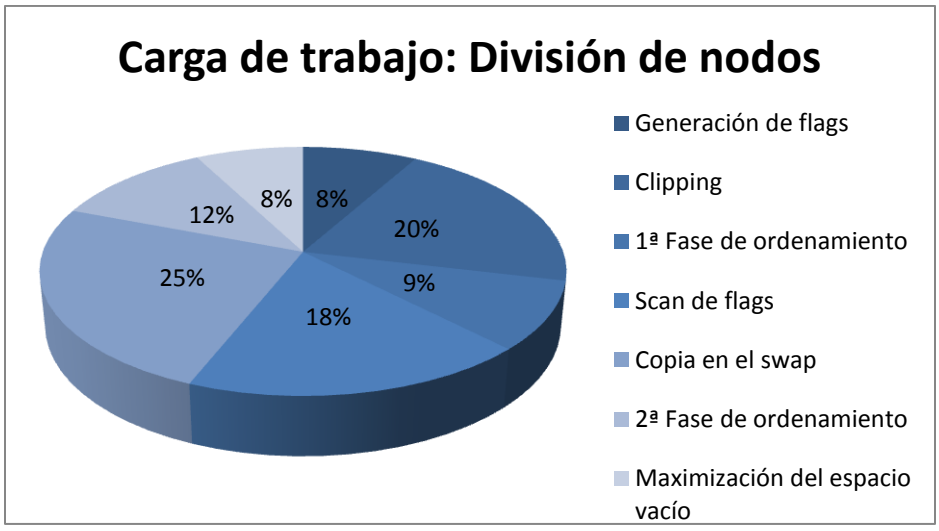
La Figura 5.10 muestra el reparto de trabajo para la etapa de selección de planos de corte. La mayor porción está cubierta para la obtención de mínimos, que se realiza mediante la operación "reduction" de THRUST (68%), seguida del scan de flags (16%), ambas operaciones costosas. Para mejorar esta fase en un trabajo futuro, sería preferible centrarse en hacerlas más eficientes.

**Figura 5.10 Carga de trabajo para las etapas de la fase de selección de planos de corte.**




Por último, la Figura 5.11 muestra el reparto para la etapa de división de nodos. La tarea que mayor tiempo consume es la copia de los eventos y estructuras asociadas (nodos al que pertenecen cada primitiva, su posición en el array de eventos, …) a la estructura de swap que sirve de entrada para la siguiente iteración del algoritmo. A continuación, los que más tiempo consumen son las tareas de clipping y scan sobre los flags, habiendo un reparto equitativo en las demás etapas.

**Figura 5.11 Carga de trabajo para las etapas de la fase de división de nodos.**



En conclusión, usar librerías con implementaciones de las operaciones “reduction”, “scan” y “sort” más eficientes que las proporcionadas por THRUST, o usar implementaciones propias, supondría una mejora en los tiempos del algoritmo. Por otra parte, de las tareas que no utilizan esas operaciones, el clipping es la que debería tener mayor prioridad a la hora de mejorar los tiempos de construcción del kd-tree.

# Capítulo 6 - Trabajos relacionados

## Construcción de Kd-tree en CPU

En 2006, Wald y Havran [WH06] hicieron un algoritmo que construía kd-trees de buena calidad secuencialmente. Este algoritmo construye recursivamente de manera "top-down" y primero en profundidad el árbol. La Figura 6.1 muestra el algoritmo: selección de mejor plano con SAH exacta (línea 3) y división del nodo (líneas 4 – 6). La entrada de este algoritmo requiere tener los eventos de las cajas de cada triángulo (eventos start son los límites mínimos de las cajas y los eventos end los máximos) ordenados (línea 11) del mismo modo que he explicado en los capítulos 3 y 4.

Los candidatos para los planos de corte que se emplean son los bordes de las cajas (SAH exacta). Tras determinar el plano de corte para el nodo, se dividen los triángulos entre el hijo izquierdo o el derecho según su localización con respecto al plano. Los que son cruzados por el plano deben ir a ambos lados mediante clipping.

**Figura 6.1 Algoritmo secuencial empleado en el artículo de Wald y Havran [WH06].**

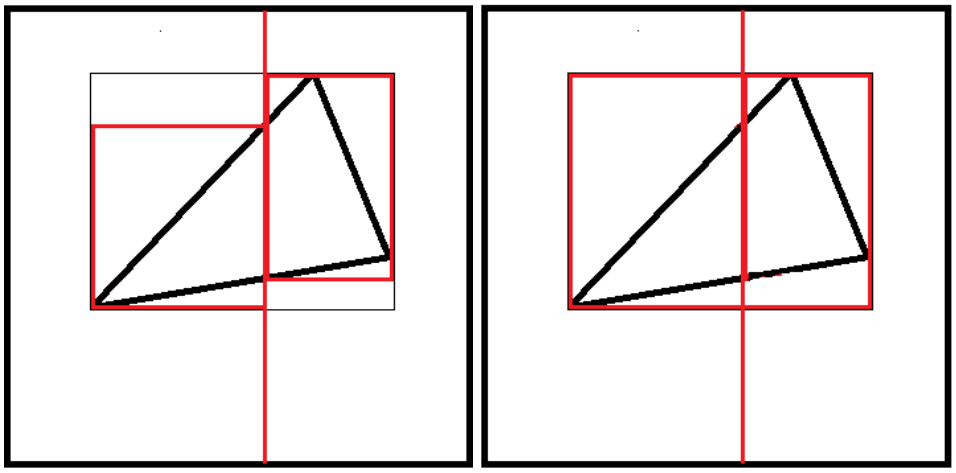
```
00 function RecBuild(triangles T, voxel V) returns node {
01     if Terminate(T, V) then
02       return new leaf node(T)
03     p = FindPlane(T, V) { Find a "good" plane p to split V}
04     (Vl, Vr) = Split V with p
05     Tl = {t ϵ T | (t ∩ Vl) ≠ Ø
06     Tr = {t ϵ T | (t ∩ Vr) ≠ Ø
07     return new node(p, RecBuild(Tl, Vl), RecBuild(Tr, Vr))
08 }
09
10 function BuildKDTree(triangles[] T) returns root node {
11     V = B(T) {start with full scene, sorting events}
12     return RecBuild(T, V)
13 }
```

En su tesis, Havran [Hav00] propuso una mejora que consiste en minimizar el tamaño de las cajas para los triángulos partidos por el eje, eliminando el espacio vacío de las AABBs. La mejora consiste en que si el rayo para ray tracing cae en un área vacía no necesita realizar intersección con las primitivas del correspondiente subárbol. También propuso la maximización del espacio vacío tras hacer el corte a un nodo. Aunque no haya triángulos cruzados por el corte, la caja del hijo izquierdo o la del derecho pueden contener huecos entre los bordes del padre y

los triángulos que han quedado a ese lado. Para las regiones vacías, se crean nodos fantasma, es decir, nodos hoja que solo tienen espacio vacío. Ejemplos de esta técnica se muestran en el Capítulo 4, (Construcción de kd-trees 2D/3D – Maximización del espacio vacío).

Recientemente, en [CKL*09], parten del algoritmo secuencial para construir una versión paralela en CPU, usando 32 núcleos con memoria compartida, haciendo un uso eficiente de ella. El problema del algoritmo es que el clipping mantiene las cajas del triángulo original, es decir, no elimina el espacio vacío como se muestra en la Figura 6.2. La imagen de la izquierda contiene un ejemplo en el que el triángulo es cortado por una recta paralela al eje Y, y las cajas que envuelven a las partes que quedan a cada lado se ajustan. Sin embargo, en [CKL*09] se hace clipping según la imagen de la derecha que mantiene los límites del padre. Según los experimentos de la tesis de Havran, ese enfoque ralentiza el algoritmo de ray tracing.

**Figura 6.2 Ejemplo de clipping que calcula la caja mínima que envuelve a la parte de cada hijo y un segundo ejemplo en el que se mantienen los límites de la figura inicial [CKL*09].**



## Primera construcción de Kd-trees completa en GPU: Zhou

En 2008, Zhou et al. [ZHWG08] desarrollaron un algoritmo que construye kd-trees completamente en GPU. Éste, clasifica los nodos según el número de triángulos que contienen para hacer un tratamiento especial en cada caso:

- Para los nodos grandes (con más de 32 triángulos), en los primeros niveles del árbol emplean la media espacial y así evitan el alto coste de calcular la SAH con tantos candidatos.

- Para los nodos pequeños (32 triángulos o menos) emplean SAH exacta. Resuelven todos los niveles de ese nodo hasta las hojas para así aprovechar la carga en memoria compartida de los triángulos, además de tener buen rendimiento por la programación a nivel de warp.

El problema de este enfoque es que el uso de la media espacial para los niveles superiores de la construcción genera malos kd-trees, afectando al resto de niveles y al renderizado por un sistema de ray tracing [CKL*09].

## Construcción de KD-Tree con "Binned SAH" en GPU

En [DPS10], enfocan la construcción de kd-trees de manera "top-down" y en anchura. Se procesan todos los nodos del mismo nivel en paralelo, y cada hilo procesa uno o más triángulos de un nodo. En el algoritmo se distinguen varias etapas que se ejecutan dependiendo del número de primitivas que tengan los nodos. El uso de medias espaciales u objetuales es un cálculo muy rápido en comparación con los cálculos para la SAH, pero la calidad del árbol es muchísimo peor. Por eso los autores del trabajo proponen emplear SAH en todos los niveles del árbol. Las diferencias de las etapas se basan en cómo se toman los candidatos para la SAH:

- SAH exacta: computar el valor del coste de la función en el lado inferior y superior de la caja de cada triángulo.
- Sampled SAH: se computa el coste de SAH para unas pocas posiciones. En concreto binned SAH toma como candidatos planos paralelos equiespaciados.

El uso de "binned SAH" es mucho más rápido porque se suele tener menos candidatos que para la SAH exacta, sin embargo, el árbol generado tiene menos calidad que usando la SAH exacta. En ese trabajo se usa "binned SAH" en los primeros y medios niveles del árbol para que el algoritmo sea más rápido, dejando sólo la SAH exacta para los niveles bajos del árbol, ya que estos tienen nodos con pocos triángulos y es rápido calcularla.

### *Computar coste*

Se emplea "binned SAH" con 32 candidatos. Se tienen dos arrays con una casilla para cada candidato: uno con la cuenta de cuántos eventos start hay entre cada pareja de planos candidatos, y otro similar para los eventos end. La manera de obtener cuántas primitivas hay a cada lado de cada plano candidato, dato necesario para computar la SAH, resulta haciendo una

suma prefija exclusiva de esos dos arrays, del comienzo al final para el de los eventos start y del final al comienzo para los eventos end.

Se comprueba también si el nodo tiene suficientes primitivas para seguir en la etapa de "binned SAH" o pasar a la de SAH exacta, o si ya no merece la pena seguir dividiendo.

### *Split*

En este punto se sabe dónde hay que cortar cada nodo y cuántas primitivas quedan a cada lado. Se reorganizan las cajas de los triángulos de cada hijo en el array de salida (entrada para la siguiente generación) de forma que ocupen posiciones contiguas de memoria. Para cada triángulo se establece en un array la marca de a qué lado debe ir, si izquierdo, derecho o ambos. Haciendo la suma prefija exclusiva se determina la posición destino dentro del array.

### *Clipping*

Emplean la técnica de clipping explicada en el capítulo 4, que mejora el árbol construido, ya que reduce el tamaño de las cajas reduciendo los espacios vacíos.

### *Implementación*

Emplean diferentes implementaciones que son eficientes según el número de triángulos y nodos que haya que tratar en cada nivel. Para ello, su programa consta de cinco implementaciones:

- *Huge stage*. En las primeras iteraciones de la ejecución los nodos contienen muchos triángulos, además de ser pocos nodos. En esta etapa varios bloques CUDA trabajan conjuntamente para tratar un solo nodo.
- *Big Stage*. Cuando hay suficientes nodos para que cada bloque se encargue de un nodo.
- *Medium stage*. En los niveles medios, los nodos con menos de 512 triángulos se asignan a un bloque CUDA que usa memoria compartida para acelerar la ejecución.
- *Small stage*. En los niveles bajos, los nodos con 32 primitivas, se asignan a un warp. De este modo, un bloque trata 4 nodos al mismo tiempo (con 4 warps).
- *Tiny stage*. Cuando los nodos tienen muy pocas primitivas un bloque trata muchos nodos simultáneamente.

## Construcción de kd-trees con SAH exacta en GPU

El trabajo de Wu, Zhao y Liu [WZL11], es al que esta memoria más se parece. Trata la construcción de los kd-trees de manera "top-down" empleando SAH exacta en todos los niveles, por lo que se generan árboles de buena calidad. Para cada iteración computan la SAH de los nodos del nivel y hacen la partición manteniendo ordenados los eventos para la siguiente generación.

### *Computación de SAH*

Se toman como candidatos para plano de corte a cada evento, es decir, los planos que contienen las caras de cada caja. La Figura 6.3 ilustra el proceso para resolver el problema de contar el número de primitivas a cada lado del plano de corte. Se tiene un array de flags en el que cada evento start escribe un 1 en su misma posición, y cada evento end un 0. Haciendo una suma prefija exclusiva se obtienen para cada evento el número de triángulos a su lado izquierdo, y con ese mismo array se puede obtener los que hay a su derecha con la fórmula de la Figura 6.4. Es decir, con un solo array se extrae la información de los triángulos que hay a cada lado. Este método supone una gran ventaja respecto a anteriores enfoques, que emplean dos arrays (uno para cada lado), ya que sólo se realiza la operación scan sobre un array cuyo tamaño es el número de eventos. Por último se usa la reducción para obtener el mínimo.

**Figura 6.3 Ejemplo de estructuras para el cómputo de la SAH, extraído de [WZL11].**

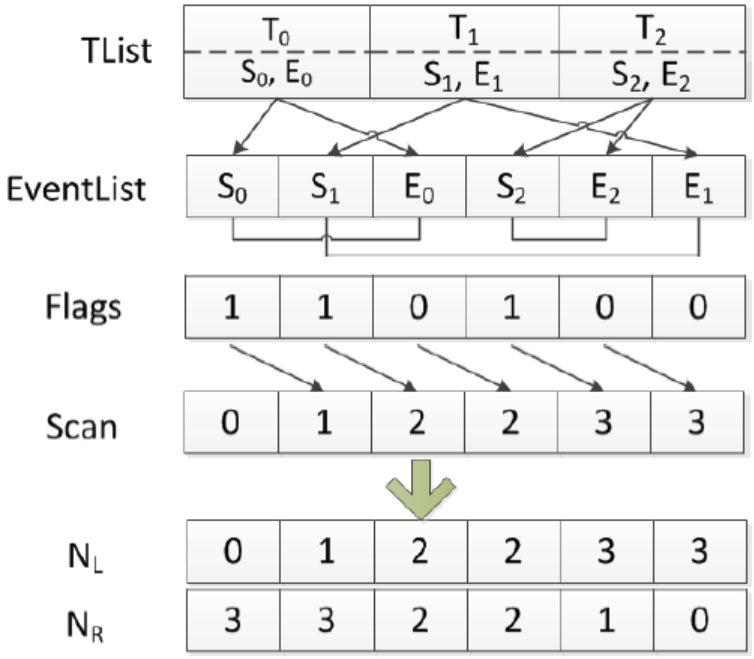


**Figura 6.4 Fórmula para hallar el número de primitivas que hay a la derecha de un plano de corte.**

$$N_R = N_T - (índice - N_L)$$

$N_R$: Número de primitivas a la derecha del plano candidato.

$N_T$: Número de primitivas del nodo.

$N_L$: Número de primitivas que se abrieron a la izquierda del plano candidato.

Índice: Índice del candidato en el nodo.

### *Partición y ordenado de eventos*

Se crean los hijos y se distribuyen los triángulos. Para los triángulos partidos por planos de corte se recalcula la caja para los hijos tras el clipping, lo que implica perder el orden de los eventos para los ejes distintos al de corte. Pero como reordenar desde cero es lento, se emplea un algoritmo de reordenamiento que es el que explico en la sección "División de nodos" del Capítulo 4 para el caso de dos dimensiones. Tras hacer el reordenamiento, se efectúa la maximización del espacio vacío.

## Acelerando algoritmos para grafos en CUDA empleando Warps

En [HKOO11], explican que los accesos a memoria irregulares en algunos algoritmos sobre grafos hacen que su implementación en CUDA no sea eficiente. Por eso proponen una plantilla para la implementación de algoritmos en CUDA centrada en el uso de warps. Asignan un conjunto de tareas a un warp. Éstas se ejecutan en orden secuencial pero cada una de ellas se ejecuta con los 32 hilos del warp en paralelo. El trabajo que se realiza en paralelo para cada tarea aprovecha la ejecución de un warp: todos los hilos ejecutan la misma instrucción al mismo tiempo sobre diferentes datos. Al serializar el conjunto de tareas total, se consigue evitar la divergencia de las bifurcaciones en el sentido de que cada warp va a ejecutar un único camino determinado por la tarea.

En la plantilla al comienzo de cada kernel se trae a memoria compartida los datos a usar de forma unificada por todos los hilos del warp, después se ejecuta el cuerpo principal del kernel serializando las tareas, intentando evitar la divergencia originada por las bifurcaciones y aislando

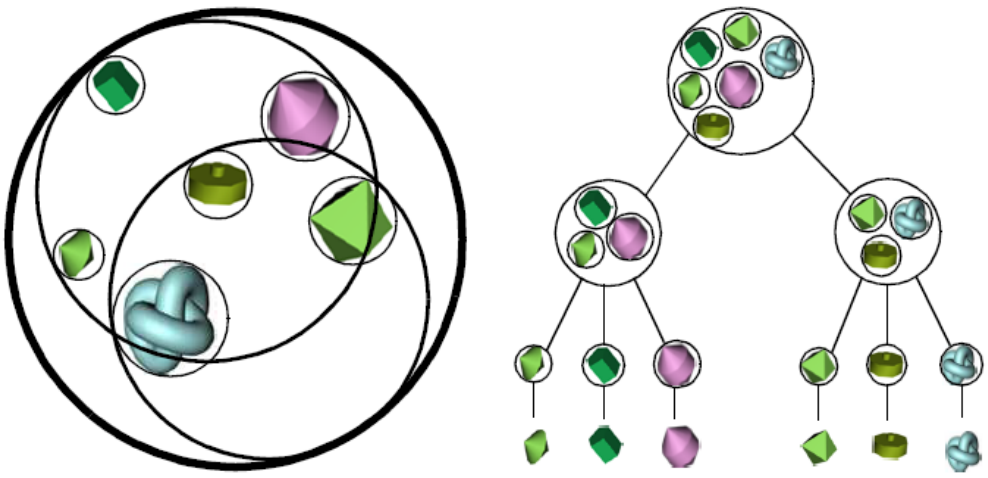
lo más posible la parte irregular de forma que todo ello gira en torno al uso eficiente de los warps.

## Bounding Volume Hierarchy (BVH)

Existe otro tipo de estructura arbórea que tiene buenos resultados para ray tracing, las BVHs. Estos árboles tienen los elementos de la escena envueltos en distintas figuras geométricas, que se denominan volúmenes, como circunferencias o esferas. Cada nodo tendrá asignado un recubrimiento similar que envuelve todos los objetos que tenga.

Una construcción "bottom-up" consiste en empezar envolviendo cada objeto con el volumen, formando los nodos hojas, que se unen luego por parejas en un nodo interno que envuelve estos dos volúmenes. Sucesivamente se van a uniendo nodos internos hasta la raíz que envuelve toda la escena. La principal diferencia con los KD-trees es que puede haber objetos que caigan en diferentes nodos internos del mismo nivel, llegando a estar en diferentes ramas del árbol. Esto tiene varias implicaciones: si un rayo interseca el hijo izquierdo de un nodo, es necesario examinar también el hijo derecho, porque puede estar ahí la intersección más cercana. Por lo tanto hay que examinar los dos hijos.

**Figura 6.5 Ejemplo de escena para agrupación de objetos con BVHs usando esferas mostrado en la tesis de Chang [Cha04].**

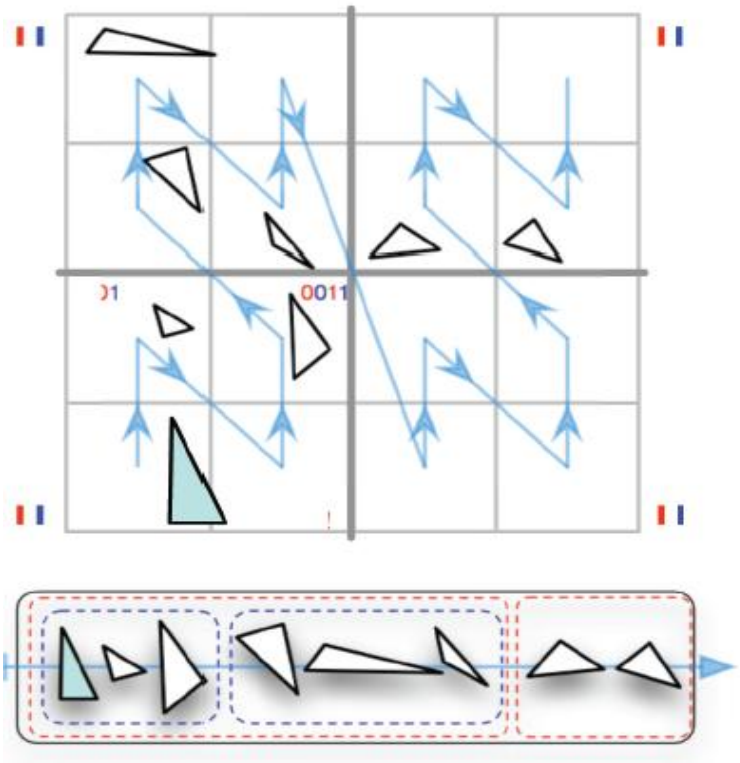


En [LGS*09] hacen una implementación en GPU para construir este tipo de estructuras. Emplean la media objetual en niveles altos del árbol para elegir los planos de corte, y el sampled SAH en los niveles más bajos.

### *Códigos de Morton*

Para obtener la media objetual emplean códigos de Morton. Una curva de Morton es una curva que rellena el espacio. Dado un punto en 3D, a partir de sus coordenadas, se le puede asignar un escalar que se llama código de Morton. En la Figura 6.6, la línea azul muestra el recorrido de la curva por los cuadrantes que forman la escena. Usando el punto central de las cajas de cada triángulo, se obtienen sus códigos de Morton, que les sitúan en un orden dentro de la curva: no es más que codificar las tres componentes del punto (X, Y y Z) en enteros de k bits y unirlos en un entero de 3k bits. Ordenando esos códigos de Morton se consigue el orden de aparición de los triángulos en la curva de Morton, lo que permite obtener la media objetual. Estas curvas se utilizan para seguir construyendo todos los niveles del árbol.

**Figura 6.6 Ejemplo del cálculo de la media objetual empleando la técnica de los códigos de Morton, extraída de [LGS*09].**



# Capítulo 7 - Conclusiones y trabajo futuro

En este trabajo se ha presentado una implementación en GPU para la construcción de kd-trees parecido al algoritmo de Wu et al. [WZL11]. Este algoritmo genera árboles de calidad para el renderizado de escenas 3D mediante ray tracing, ya que emplea la heurística SAH exacta en todos los nodos del árbol para determinar los planos de corte.

Entre las diferencias que aporto y que mejoran el algoritmo original, he reducido el tamaño del array sobre el que calcular la mínima SAH. Ya que el cálculo de la SAH se hace por primitiva y que cada una aporta tiene 6 candidatos, el array global queda reducido a una sexta parte. Por otra parte, en mi implementación se utiliza la operación "reduction" de la librería THRUST para calcular la mínima SAH, mientras que Wu et al. implementan su propia operación "reduction" adaptada a la construcción de kd-trees (no sólo encuentran el mínimo valor de SAH sino que también el evento asociado). Según Wu, su implementación mejora la reducción de THRUST.

Como posibles mejoras se podría reducir el uso de memoria en GPU, reutilizando estructuras auxiliares en diferentes partes del algoritmo independientes entre sí, o reduciendo el tamaño de las estructuras para los nodos. Éstas vienen determinadas por estructuras para árboles completos, aunque el algoritmo pueda generar árboles no completos. Por ejemplo, la Figura 7.1 a) contiene espacio para los nodos de los tres primeros niveles de un árbol binario completo. Sin embargo, el nodo 1 es un nodo hoja, y aunque sus descendientes (los nodos 3 y 4) no existen, están ocupando espacio en memoria. Es decir, el árbol generado por el algoritmo no utiliza las posiciones del array que no necesita. La mejora consiste en reducir el tamaño de ese array, tal y como se muestra en la Figura 7.1 b), conteniendo sólo los datos útiles.

En cuanto al rendimiento, reducir el tamaño de los arrays que sirven de entradas de las operaciones "scan" y "reduction" es clave porque en ellas recae la mayor carga de trabajo del algoritmo. Además, podría usarse librerías con mejor rendimiento que THRUST (por ejemplo CUDPP, que no he empleado en este trabajo por no haber disponible una versión compatible con la última versión del SDK Toolkit de CUDA), o realizar una implementación propia que mejore ambas operaciones.

**Figura 7.1 a) Estructura que reserva espacio para un árbol completo de 3 niveles. b) Estructura que reserva el espacio que requiere un dato para un árbol de 3 niveles.**

a)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| d_nodeNumSegments | 4 | 1 | 3 | | | 1 | 2 |

b)

| | 0 | 1 | 2 | 5 | 6 |
|---|---|---|---|---|---|
| d_nodeNumSegments | 4 | 1 | 3 | 1 | 2 |

Una siguiente fase de desarrollo consistiría en crear implementaciones adaptadas a diferentes situaciones durante la construcción del árbol. Por ejemplo, los nodos con pocos triángulos (alrededor de 32) pueden generar varios niveles. En el enfoque actual, el algoritmo continúa ejecutando las iteraciones que completarían esos niveles, ejecutando todos los kernels del algoritmo, con el movimiento de datos de memoria correspondiente. Sin embargo, otro enfoque más eficiente consistiría en completar esos niveles con un solo kernel, en el que cada warp trata un nodo, reutilizando los datos leídos de memoria. Esto sería posible porque el nodo tiene pocos triángulos, dando lugar a operaciones scan y reduction de bajo coste, implementadas dentro del kernel. Además, los datos leídos de memoria se pueden guardar en memoria compartida para su reutilización.

Por último, como paso posterior a este trabajo, podría implementarse también un algoritmo de “ray tracing” en GPU para el renderizado de escenas en tiempo real, que haga uso de los árboles construidos.

# Referencias


[GS87] Goldsmith, J., and Salmon, J. 1987. Automatic creation of object hierarchies for ray tracing. IEEE Comput. Graph. Appl. 7 (May). 14-20.

[Hav00] Havran, V. 2000. Heuristic Ray Shooting Algorithms. Ph.d. thesis, Department of Computer Science and Engiinering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[MB90] MacDonald, D., and Booth, K. 1990. Heuristics for ray tracing using space subdivision. Vis. Comput. 6 (May), 153-166.

[Cha04] Allen Yao-Hung Chang, May 22, 2004. Theoretical Doctoral Dissertation Theoretical and experimental aspects of ray shooting Polytechnic University Brooklyn, NY, USA 2004.

[WH06] Wald, I., and Havran, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(n log n). Symposium on Interactive Ray Tracing 0, 61-69.

[PGS07] Popov, S., Günther, J., Seidel, H-P, and Slusallek, P. 2007. KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum, 26(3)

[DPS10] Danilewski, P., Popov, S., and Slusallek, P. Binned SAH Kd-Tree Construction on a GPU. Saarland University, June 2010.

[CKL*09] Choi B., Komuravelli R., Lu V., Sung H., Bocchino R. L., Adve S. V., Hart J. C.: Parallel SAH k-d Tree Construction for Fast Dynamic Scene Ray Tracing. Tech. rep., University of Illinois, 2009.

[LGS*09] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. 2009. Fast bvh construction on gpus. Comput. Graph. Forum 28, 2, 375-384.

[ZHWG08] Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Real-time kd-tree construction on graphics hardware. ACM Transactions on Graphics, 126.

[HKOO11] Hong S., Kyun Kim S., Oguntebi T., Olukotun K. 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, 267-276.

[WZL11] Wu, Z., Zhao, F., Liu, X. 2011. Sah KD-Tree construction on GPU. HPG '11 Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics Pages 71-78